21262-1-1

APPENDIX

```
Introduction: This is the source code for the ExpertFinder system,
a mixture of HTML/Javascript and Macintosh Common Lisp.
Copyright (C) 2000,  John Sotos          All Rights Reserved.

There are multiple source code files combined into this single file.
The borders between the original files are delimited by "@@file" lines.


@@file $boot

(defparameter *app-packages*
  '((:name "TOKENIZER" :nickname "TK" :var *tk-package*)
    ))

(defparameter *app-logical-drive* "C") ; used by PC only
(defparameter *app-logical-directories*
  '(
    ("source"
     #+mcl "HD20a:ayudame"
     #-mcl "pubmed\\")
    ("searches"
     #+mcl "HD20a:ayudame:searches"
     #-mcl "pubmed\\searches")
    ))

(defparameter *app-files-to-load*
  '(
    ;;; Load macros not specific to this application
    ("source:xserve-macros")
    ("source:lisp-macros")

    ;;; Load code not specific to this application
    ("source:platform")                 ; platform-specific functions
    ("source:serv_mcl")                 ; platform-specific server interface
code
    ("source:xserve")                   ; server-interface code. Some app-
specific defs.

    ;;; Load code specific to this application
    ("source:analyzer")
    ("source:places")
    ("source:pickfield")                ; mostly defines new classes

    ("source:rules")
    ("source:places-find")
    ("source:noder")

    ("source:cgi-fns")
    ("source:masterpick")
    ("source:parse-paper")

    ;;; Load data specific to this application
    ("source:countries")
    ;("source:states")     -- loaded by countries file
    ;("source:city_inst") -- loaded by states file
    ))

(defparameter *app-post-load-eval* '(initialize-searches))
(defparameter *app-post-load-msg* "Entry point is (tally)")

#+mcl (proclaim '(optimize (debug 3)))
```

```
#+mcl (let ((save-stuff-p #+production-system nil #-production-system t))
        (setf *save-definitions*        save-stuff-p
              *save-doc-strings*        save-stuff-p
              *save-local-symbols*      save-stuff-p
              *record-source-file*      save-stuff-p
              *fasl-save-local-symbols* nil))

;;; ---------------------------------------------------------------------------
-----------|
;;; ---------------------------------------------------------------------------
-----------|
;;; Generic boot-up code follows.
;;; ---------------------------------------------------------------------------
-----------|
;;; ---------------------------------------------------------------------------
-----------|
;;; PACKAGES

;;; These are some conveniences I like.

(rename-package 'common-lisp-user 'common-lisp-user ; add new nickname
                (adjoin "U" (package-nicknames 'common-lisp-user) :test
#'string=))

(rename-package 'common-lisp   'common-lisp         ; add new nickname
                (cons 'lisp (package-nicknames 'common-lisp  )))

(or (find-package "BOOT") (make-package "BOOT" :use '("U" "LISP")))

;;; Define user's packages

(defmacro boot::define-packages (package-spec-symbol)
  (flet ((define1 (package-spec)
           (let ((name     (getf package-spec :name nil))
                 (nickname (getf package-spec :nickname nil))
                 (var      (getf package-spec :var nil))
                 (form     nil))
             (setq form
                   `(or (find-package ,name)
                        (make-package ,name
                                      :use '("U" "LISP")
                                      :nicknames '(,nickname))))
             (when var
               (setq form (list 'defparameter var form)))
             (values form))))
    (cons 'progn
          (mapcar #'define1 (symbol-value package-spec-symbol)))))

(boot::define-packages *app-packages*)

;;; ---------------------------------------------------------------------------
-----------|
;;; Universal utilities

(defun dbp (&rest args)
  (format t "~%;;;~{ ~S~}" args)
  (values))

;;; ---------------------------------------------------------------------------
-----------|
;;; Pathname fns
```

```
(flet ((fullpath (logical-path-string)
         (let* ((colonpos   (or (position #\: logical-path-string :test
#'char=)
                                (error "No colon in logical path string ~S"
                                       logical-path-string)))
                (dotpos     (position #\. logical-path-string :test #'char=
:from-end t))
                (dir        (subseq logical-path-string 0 colonpos))
                (extension  (and dotpos
                                 (subseq logical-path-string (1+ dotpos))))
                (star-exten (and extension
                                 (string= extension ".*")))
                (directory  (or (second (find dir *app-logical-directories*
                                              :key #'first :test #'string-
equal))
                                (error "Cannot find logical directory: ~S"
dir)))
                (name       (subseq logical-path-string (1+ colonpos)
                                    (or dotpos (length logical-path-
string))))
                )
           #+mcl
           (cond
            (star-exten
             (make-pathname :directory directory :name name :type "*"))
            (extension
             (make-pathname :directory directory :name name :type extension))
            (t
             (make-pathname :directory directory :name name)))
           #-mcl
           ; (make-pathname :device "C" :directory "pubmed\\searches" :name
"xyz.html")
           (make-pathname :device *app-logical-drive*
                          :directory directory
                          :name name
                          :type "*"))))

  (defun find-latest-path (logical-pathstring)
    "Loads the latest version of a file.  Latest = alphabetically last.
     Argument should be a string.
     Filenames should have form 'name.ext'"
    ;; < merge-pathnames > yields "name.*" in MCL, at least.
    (let* ((pathname (fullpath logical-pathstring))
           (files (directory (merge-pathnames #.(make-pathname) pathname))))
      (if files
        (first (sort files #'string-greaterp :key #'namestring))
        (error "Missing file ~S" pathname))))

  (defun find-path (logical-pathstring)
    (fullpath logical-pathstring)))

(defun boot::load-logical-path-file (logicalpathname &key (printp t))
  (let ((path (find-latest-path logicalpathname)))
    (when printp
      (dbp path))
    (load path)))


;;; ---------------------------------------------------------------------
-----------|
```

```
;;; Load files

(map nil #'boot::load-logical-path-file (mapcar #'first *app-files-to-load*))

(eval *app-post-load-eval*)
(format t "~&Done loading.~%~A" *app-post-load-msg*)

@@file analyzer

(defstruct author
  name
  allpapers
  firstauthorpapers
  lastauthorpapers
  (score 0)
  )

(flet ((p (papers)
         (if papers
            (subseq (format nil "~{,~A~}" (mapcar 'paper-pmid papers))
              1)
            "")))
  (defmethod print-object ((x author) s)
     (format s "<AUTHOR name='~A' allpapers=~A firstauthorpapers=~A>"
       (author-name x)
       (p (author-allpapers x))
       (p (author-firstauthorpapers x)))))

(defstruct paper
  ;; Primary data
  pmid
  title
  authornames
  lastauthor
  pt              ; publication type
  address
  year
  country
  authorcount
  ;;
  ;; Derived
  ;;
  allauthors
  (score 0)
  rules                                  ; rules -- for places we deduce from
address
  leafplaces                             ; places -- deduced from geographic
hierarchy
  leafplacenodes
  email
  )

(defvar *tally*)
(defparameter *tallies* nil)

(defstruct tally
  key
  authordata
  paperdata
  searchparms ; used?
  filename
```

```
  nodes
  topnode
  ;;
  ;; These are pickfields.
  ;;
  wts
  geo
  srt
  fmt
  max
  ;;
  ;; This is a pseudo-pickfield
  ;; (is a link parameter, but not a pickfield)
  ;;
  (sta 1)
  )

(defvar *line-buffer*)

(defun create-tally (&key name date time utc papers)
  ;; This function is called from within a database file, which undergoes a
Lisp LOAD.
  ;; The only way to return a value to the calling function is with a throw.
  (declare (ignore date time utc))
   (let ((tally (make-tally
                 :key        (length *tallies*)
                 :filename   name
                 :authordata nil
                 :paperdata  papers)))
     (setf (tally-authordata tally)
           (make-hash-table :test #'equal
                            :size (* 2 (reduce #'+ papers :key #'paper-
authorcount))))
     (throw 'tally-created tally)))

(defun lxl-post (tally)
  (dovec (paper i (tally-paperdata tally))
    (dolist (authorname (paper-authornames paper))
      (process-author tally paper authorname))
    (push paper (author-firstauthorpapers
                 (get-author-rec tally (first (paper-authornames paper)))))
    (push paper (author-lastauthorpapers
                 (get-author-rec tally (paper-lastauthor paper)))))
  (map nil #'score-paper   (tally-paperdata tally))
  (map nil #'score-authors (tally-paperdata tally)))

(defun get-author-rec (tally authorname)
  ;; Returns
  (or (gethash authorname (tally-authordata tally) nil)
      (error "No author with name ~S" authorname)))

(defun tally-from-tallykey (key)
  "Returns NIL or a tally object"
  (find key *tallies* :test #'= :key #'tally-key))

(defun tally-new-paper (tally)
   (let ((paper (make-paper)))
      (vector-push-extend paper (tally-paperdata tally))
      (values paper)))

(defun score-paper (paper)
```

```
  ;; Important that every paper gets at least one point.
  ;; Otherwise it's possible for a node to exist and have zero points,
  ;; which could lead to divide-by-zero errors in the score bar.
  (let ((pt (mapcar #'string-upcase (paper-pt paper))) ; just in case
        (bonus 0))
    (cond ((member "EDITORIAL" pt :test #'string=)
           (incf bonus 25))
          ((some #'(lambda (pubtype) (search "GUIDELINE" pubtype :test
#'char=)) pt)
           (incf bonus 30))
          ((notany #'(lambda (pubtype) (search "REVIEW" pubtype :test
#'char=)) pt)
           (incf bonus 1))
          ((member "REVIEW OF REPORTED CASES" pt :test #'string=)
           (incf bonus 10))
          ((member "REVIEW, MULTICASE" pt :test #'string=)
           (incf bonus 10))
          ((member "REVIEW LITERATURE" pt :test #'string=)
           (incf bonus 8))
          ((member "REVIEW" pt :test #'string=)
           (incf bonus 5))
          (t
           (dbp pt)))
    (setf (paper-score paper) bonus)))

(defun score-authors (paper)
   (let* ((authors (paper-allauthors paper))
          (n-authors (length authors))
          (a1 (first authors))
          (an (first (last authors)))
          (delta (+ 1 ; at least one point for every paper
                    (paper-score paper))))
      (dolist (au (paper-allauthors paper))
         (incf (author-score au) delta))
      (incf (author-score a1) (case n-authors (1 2) (2 0) (3 1) (4 2) (t 3)))
      (when (>= n-authors 3)
         (incf (author-score an) (case n-authors          (3 1) (4 1) (t
2))))
      ))

(defun process-author (tally paper authorname)
   (let* ((authorhash (tally-authordata tally))
          (author-rec (or (gethash authorname authorhash nil)
                          (setf (gethash authorname authorhash)
                                (make-author
                                 :name authorname
                                 :allpapers nil
                                 :firstauthorpapers nil)))))
      (push paper (author-allpapers author-rec))
      (push author-rec (paper-allauthors paper))
      (values author-rec)))

(defun author-addresses (author)
   (let ((addresses nil))
      (dolist (paper (author-firstauthorpapers author))
         (push (paper-address paper) addresses))
      (values addresses)))

(defparameter *pubmed-script*
  (format nil "~&<script>~
```

```
~%pmurl='http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=PubMed&cmd=Retrieve
&dopt=Abstract&list_uids=';~
                ~%function pm (id) {window.open(pmurl+id,'Experts')}~
                ~%</script>"))

(defun find-search (filename)
  (find-path (format nil "searches:~A" filename)))

(defun make-outpathname (inpathname)
  (find-path (format nil "searches:~A.txt" (pathname-name inpathname))))

(defun read-html-and-emit-lxl (filename inpathname outpathname)
  (with-open-file (infile inpathname :direction :input)
    (with-open-file (outfile outpathname :direction :output
                             :if-exists :supersede)
      (format outfile "(create-tally :name ~S :date ~S :time ~S :utc ~S
:papers (vector"
              filename "mmddyy" "hhmmss" (get-universal-time))
      (catch 'end-of-file
        (loop
          (find-paper-start infile)
          (parse-paper infile outfile)))
      (format outfile "))"))
    (values)))

(defun set-paper-locations (tally)
  (dovec (paper i (tally-paperdata tally))
    (multiple-value-bind (rules email)
                         (locate (paper-address paper))
      (setf (paper-email paper) email)
      (setf (paper-leafplaces paper)
            (find-leafplaces nil #'certainrule-place
                             (setf (paper-rules paper) rules))))))

(flet ((okp (candidate otherplace)
         (or (eql candidate otherplace)
             (place2-is-or-isin-place1 otherplace candidate))))
  (defun find-leafplaces (leaves place-from-place-spec-fn place-specs)
    (dolist (place-spec place-specs)
      (let ((place (funcall place-from-place-spec-fn place-spec)))
        (unless (find place place-specs :key place-from-place-spec-fn :test-
not #'okp)
          (pushnew place leaves))))
    ;(when (and (cdr leaves)
    ;           (> (count-if #'institution-p leaves) 1))
    ;  (setq leaves (remove-if #'institution1-is-part-of-institution2
leaves)))
    (values leaves)))

(flet ((okp (candidate otherplace)
         (or (eql candidate otherplace)
             (not (place2-is-or-isin-place1 candidate otherplace)))))
  (defun find-leafplaces (leaves place-from-place-spec-fn place-specs)
    (dolist (place-spec place-specs)
      (let ((place (funcall place-from-place-spec-fn place-spec)))
        (unless (find place place-specs :key place-from-place-spec-fn :test-
not #'okp)
          (pushnew place leaves))))
    ;;
```

```
    ;; When one institution is part of another, they can both show up as
leafplaces.
    ;; Here, we remove the larger institution.
    ;;
    (when (and (cdr leaves)
               (> (count-if #'institution-p leaves) 1))
      (let ((leaves2 nil))
        (dolist (leaf leaves)
          (unless (find leaf leaves :test #'institution2-is-part-of-
institution1)
            (push leaf leaves2)))
        (setq leaves leaves2)))
    (values leaves)))

(defun find-leafplaces-for-papers (papers)
  (let ((leaves nil))
    (dolist (paper papers)
      (setq leaves (find-leafplaces leaves
                                    #'identity
                                    (paper-leafplaces paper)))
      ;(FORMAT T "~%~%~%~A~%   ~S" (paper-address paper) (mapcar #'place-key
leaves))
      )
    ;; nreverse preserves the order in which the papers were presented
    (values (nreverse leaves))))

;;; ---------------------------------------------------------------------------
-----------|

(defstruct search
  pathname filename name description tallied-p (n-authors "?") (n-papers
"?"))

(defvar *searches*)
(defparameter *search-descriptions*
  '(("brocc" . "Broccoli")
    ("sas500" . "Sleep apnea and surgery")
    ("sa" . "Sleep apnea")))

(defun initialize-searches ()
  (setq *searches* nil)
  (dolist (path (directory (find-path "searches:*.html")))
    (add-search :pathname path)))

(defun add-search (&key pathname filename)
  (assert (and (or pathname filename)
               (not (and pathname filename)))
          nil "Specify either pathname or filename to ADD-SEARCH.")
  (if pathname
    (setq filename (format nil "~A.html" (pathname-name pathname)))
    (setq pathname (first (directory (find-path "searches:*.html")))))
  (let ((name (pathname-name pathname)))
    (assert (every #'alphanumericp name) nil
            "Bad search name ~S -- only letters and digits allowed." name)
    (push (make-search :pathname pathname
                       :filename filename
                       :name name
                       :description (or (cdr (find name *search-descriptions*
                                                   :key #'car
                                                   :test #'string-equal))
                                        "--none--"))
```

```
          *searches*)))

(defun search-tally (search)
  (find (search-filename search) *tallies* :test #'string-equal :key #'tally-
filename))

(defun search-name-from-tally-filename (filename)
  (search-name (find filename *searches* :key #'search-filename :test
#'string-equal)))

(defun search-load (search)
  (tally (search-filename search))
  (setf (search-tallied-p search) t))

(defun search-load-all ()
  (dolist (search *searches*)
    (unless (search-tallied-p search)
      (search-load search))))

(defun search-unload (search)
  (dbp "SEARCH-UNLOAD IS STUB")
  (setf (search-tallied-p search) nil))

(defun search-unload-all ()
  (tally :erase)
  (dolist (search *searches*)
    (setf (search-tallied-p search) nil)))

(defun tally (filename &key show erase all parse)
  ;; Exists only so we can call from command line, without huge return-value.
  (let* ((inpathname (find-search filename))
         (outpathname (make-outpathname inpathname)))
    (cond
     (show  (format t "~{~%~S~}" (directory (find-path "searches:*.html"))))
     ;;
     (erase (setq *tallies* nil)
            'erased)
     ;;
     (all   (dolist (path (directory (find-path "searches:*.html")))
              (tally (format nil "~A.html" (pathname-name path)))))
     ;;
     ((find filename *tallies* :key #'tally-filename :test #'string-equal))
     ;;
     (parse (time (read-html-and-emit-lxl filename inpathname outpathname)))
     ;;
     (t     (tally1 filename outpathname)))))

(defun tally1 (filename outpath)
  (unless (probe-file outpath)
    (tally filename :parse t))
  (let ((tally (catch 'tally-created (time (load outpath)))))
    (time (lxl-post tally))
    (time (set-paper-locations tally))
    (time (make-nodes tally))
    (time (score-nodes tally))
    (pushnew tally *tallies*)
    (values tally)))

@@file cgi-fns
```

```
(defun html-image-tag (thread src &key (border nil) (align nil)
                                       (width nil) (height nil))
  (formatt thread "<image src='imj/~A'" src)
  (when border
    (formatt thread " border='~A'" border))
  (when align
    (formatt thread " align='~A'" align))
  (when width
    (formatt thread " width='~A'" width))
  (when height
    (formatt thread " height='~A'" height))
  (princt ">" thread))

(defun summary (thread tally)
  (formatt thread "~%<br>~D papers.~%<br>~D authors.~%<br>"
           (length (tally-paperdata tally))
           (hash-table-count (tally-authordata tally))))
;;; ---------------------------------------------------------------------------
-----------|

(defun node-aside (stream node)
  (format stream "   ~S" (length (node-papers node)))
  (when (node-papers? node)
    (format stream "--~S" (+ (length (node-papers node))
                             (length (node-papers? node)))))
  (format stream "p~S" (node-score node))
  (when (plusp (node-score? node))
    (format stream "--~S" (node-total-score node)))
  (princ "s" stream))

;;; ---------------------------------------------------------------------------
-----------|

(defmethod node-print-name (node)
  (let* ((name (place-name (node-place node)))
         (pos (position #\$ name :test #'char=)))
    (if pos
      (format nil "~A (~A)"
              (subseq name 0 pos)
              (string-upcase (subseq name (1+ pos))))
      name)))

#|
(defmethod node-print-name ((node authornode))
  (author-name (authornode-place node)))

(defmethod node-print-name ((node papernode))
  (paper-title (papernode-place node)))

|#

;;; ---------------------------------------------------------------------------
-----------|

(defcgifn main ()
  (main-html thread))

(defparameter *main-javascript*
  "<script>function su(ac,ta) { f=document.forms[0];
f.elements['ACT'].value=ac;f.elements['TGT'].value=ta; f.submit();
}</script>"
```

```
  )

(defhtmfn main-html (thread) :props nil
  (with-new-page (thread :title "Welcome to S.O.T.O.S."
                         :head *main-javascript*)
    (with-thread-output (stream thread)
      (cgi-form-start thread 'searchaction)
      (cgi-form-hidden thread
                       'act ""
                       'tgt "")
      (format stream "~%<table cellpadding='5'><tr>~
                      <th>Search Name</th>~
                      <th>Description</th>~
                      <th>Status</th>~
                      <th>Action</th>~
                      <th>             </th>~
                      <th>Load Mgmt</th></tr")
      (dolist (search *searches*)
        (format stream "~%<tr>~@{<td>~A</td>~}</tr>"
                (search-name search)
                (search-description search)
                (if (search-tallied-p search) "Loaded" "Not loaded")
                (if (search-tallied-p search)
                  (format nil
                          "<input type='button' value='View'
onclick='su(~S,~S)'>"
                          "V" (search-name search))
                  (format nil
                          "<input type='button' value='Load + View'
onclick='su(~S,~S)'>"
                          "LV" (search-name search)))
                ""
                (if (search-tallied-p search)
                  (format nil
                          "<input type='button' value='Unload'
onclick='su(~S,~S)'>"
                          "U" (search-name search))
                  (format nil
                          "<input type='button' value='Load'
onclick='su(~S,~S)'>"
                          "L" (search-name search)))))
      (format stream "~%<tr>~@{<td>~A</td>~}<td>~
                      <p><input type='button' value='Unload all'
onclick='su(\"UA\",0)'>~
                      <p><input type='button' value='Load all'
onclick='su(\"LA\",0)'>~
                      </tr>"
              "" "" "" "" "")
      (format stream "~%</table></form>"))))

;;; ---------------------------------------------------------------------------
-----------|

(defcgifn searchaction (act tgt)
  (assert (member act '("L" "V" "U" "LV" "LA" "UA") :test #'string-equal) nil
          "Illegal action ~S" act)
  (let ((target (or (string-equal act "LA")
                    (string-equal act "UA")
                    (find tgt *searches* :key #'search-name :test #'string-
equal))))
    (cond
```

```
      ((string-equal act "L")    (search-load target)
                                 (main-html thread))
      ((string-equal act "V")    (search-view-html thread target))
      ((string-equal act "U")    (search-unload target)
                                 (main-html thread))
      ((string-equal act "LV")   (search-load target)
                                 (search-view-html thread target))
      ((string-equal act "LA")   (search-load-all)
                                 (main-html thread))
      ((string-equal act "UA")   (search-unload-all)
                                 (main-html thread)))))

(defcgifn searchhome (id)
  (searchaction thread :act "V" :tgt id))

;;; ---------------------------------------------------------------------------
-----------|

#|
(defhtmfn error-page (thread title fmt$ fmt-args) :props nil
  (with-new-page (thread :title (format nil "Error - ~A" title))
    (apply #'formatt-fn thread fmt$ fmt-args)))
|#

(defun escape-author-name (name)
  ;; no unescape function is needed, because that is taken care of routinely
when
  ;; arguments are accepted from the web browser.
  (substitute #\+ #\Space name :test #'char=))

;;; ---------------------------------------------------------------------------
-----------|

(defhtmfn dummy-thanks (thread filename) :props nil
  (with-new-page (thread :title "Thanks!")
    (formatt thread "~%~%You submitted: ~S~%~%Thank you." filename)))

(defcgifn say_hello (repetition) :props nil
  (with-new-page (thread :title "Thanks!")
    (dotimes (i (read-from-string repetition))
      (formatt thread "~%<br>Hello world!"))
    (formatt thread "~%<p><a href='javascript:history.back()'>Go
Back</a><p>")
    (cgi-anchor-with-text thread "Show tallies" 'main))) ; show_tallies

(defun make-dummy-thread ()
  (make-thread :session (make-a-session)
               :accumulator-stream (open-text-window)))


@@file city_inst

#|
Invisible cities conclude about the country, not the about the city.
(Actually, they conclude about whatever the 2step key is.)
So, it makes no sense to have a rule like:
    :dx-helpx "Athens" -AND- co gr
Because this rule would have its conclusion in its antecedent.
So these sorts of things tend to get expressed as:
    :dx-helpx "Athens" -NOT- co us
This is great, but a problem arises when there is a pair of rules
```

```
that are the same except for the -NOT- / -AND- :
    :dx-helpx "Athens" -NOT- co us
    :dx-helpx "Athens" -AND- co us
If the country is not established before the :dx-helpx rules are
tested, then the out come can depend on which rule is tested
first! The best approach is to pick the less reliable rule and make
it :sus-helpx
    :sus-helpx "Athens" -NOT- co us ; usually true
    :dx-helpx  "Athens" -AND- co us ; always true

Note that the above does not apply to cities in the United Kingdom,
Canada, or Ireland.  There, we are trying to make a diagnosis at the
city and institution level.  So it does make sense to say:
     :dx-helpx "Dublin" -AND- co ie
|#

(map nil #'make-a-city
 '((co de "Aachen" :dx1 :invisible)
   (co uk "Aberdeen$uk" :dx-helpx "Aberdeen" -AND- co uk :dx-all "Aberdeen
Scotland")
   (co et "Addis Ababa" :dx2 :invisible)
   (co au "Adelaide" :dx1 :invisible)
   (st oh "Akron" :dx-helpx "Rootstown" -AND- st oh)
   (st ny "Albany" :dx1)
   (st nm "Albuquerque" :dx1)
   (co jo "Amman" :dx1 :invisible)
   (co nl "Amsterdam" :dx-helpx "Amsterdam" -NOT- co us :invisible)
   (co tr "Ankara" :dx1 :invisible)
   (st mi "Ann Arbor" :dx2)
   (st ga "Atlanta" :dx1)
   (co gr "Athens$gr" :sus-helpx "Athens" -NOT- co us :invisible)
   (st ga "Athens$ga" :dx-helpx  "Athens" -AND- st ga)
   (co nz "Auckland" :dx1 :invisible)
   (st ga "Augusta$ga" :dx-helpx "Augusta" -AND- st ga)
   (co de "Bad Nauheim" :dx2 :invisible) ; Max Planck
   (co in "Bangalore" :dx1 :invisible)
   (st md "Baltimore" :dx1)
   (co th "Bangkok" :dx1 :invisible)
   (co es "Barcelona" :dx1 :invisible)
   (co ch "Basel" :dx1 :invisible)
   (co uk "Bath" :dx-helpx "Bath" -AND- co uk)
   (co il "Beer-Sheva" :dx1 :invisible)
   (co cn "Beijing" :dx1 :invisible)
   (co uk "Belfast" :dx1)
   (st ca "Berkeley" :sus "Berkeley" :dx-helpx "Berkeley" -AND- st ca)
   (co de "Berlin" :dx1 :invisible) ; ignore New Hampshire
   (co ch "Bern" :sus "Bern" :invisible) ; just seems like there's more
   (st md "Bethesda" :dx1)
   (st al "Birmingham$al" :dx-helpx "Birmingham" -AND- co us)
   (co uk "Birmingham$uk" :dx-helpx "Birmingham" -AND- co uk)
   (co it "Bologna" :dx1 :invisible)
   (st ma "Boston" :dx1 :dx-helpx "Chestnut Hill" -AND- st ma) ; for BI-
Deaconness
   (co sk "Bratislava" :dx1 :invisible)
   (co au "Brisbane" :dx1 :invisible)
   (st ny "Buffalo" :dx1)
   (co ar "Buenos Aires" :dx2 :invisible)
   (co in "Calcutta" :dx1 :invisible)
   (st ab "Calgary" :dx1)
   (co uk "Cambridge$uk" :sus-helpx "Cambridge" -AND- co uk)
```

```
   (st ma "Cambridge$ma" :sus-helpx "Cambridge" -AND- co us  :dx-helpx
"Cambridge" -AND- st ma)
   (co uk "Cardiff" :dx1)
   (st sc "Charleston" :dx-helpx "Charleston" -AND- st sc)
   (st va "Charlottesville" :dx1)
   (co jp "Chiba" :sus "Chiba" :invisible)
   (st il "Chicago" :dx1
       :dx "Evanston"
       :dx-helpx "Maywood" -AND- st il) ; Loyola
   (st oh "Cincinnati" :dx1)
   (st oh "Cleveland" :dx-helpx "Cleveland" -AND- st oh) ; Cleveland Clinic
Fla.
   (co de "Cologne" :dx1 :dx "Koln" :invisible)
   (st mo "Columbia$mo" :dx-helpx "Columbia" -AND- st mo)
   (co dk "Copenhagen" :dx1 :invisible)
   (co ie "Cork" :dx1)
   (co bd "Dacca" :dx1 :dx "Dhaka" :invisible)
   (st tx "Dallas" :dx1)
   (co in "Delhi" :dx1 :invisible) ; don't care vs. New Delhi
   (st co "Denver" :dx1)
   (st mi "Detroit" :dx1)
   (co de "Dresden$de" :sus "Dresden" :invisible)
   (co ie "Dublin" :sus "Dublin" :dx-all "Dublin Ireland") ; there must be
others
   (co uk "Dundee" :dx1)
   (co za "Durban" :dx1 :invisible)
   (co de "Dusseldorf" :dx1 :invisible)
   (st wi "Eau Claire" :dx-helpx "Eau Claire" -AND- co us)
   (co uk "Edinburgh" :dx-helpx "Edinburgh" -AND- co uk)
   (st ab "Edmonton" :dx1)
   (st ct "Farmington$ct" :dx-helpx "Farmington" -AND- st ct)
   (co it "Florence$it" :dx "Firenze" :dx-helpx "Florence" -NOT- co us
:invisible)
   (st nc "Fort Bragg" :dx-helpx "Fort Bragg" -AND- st nc)
   (st wa "Fort Lewis" :dx-helpx "Fort Lewis" -AND- st wa)
   (st tx "Fort Sam Houston" :dx2)
   (co de "Frankfurt$de" :sus-helpx "Frankfurt" -NOT- co us :invisible)
   (st ky "Frankfurt$us" :dx-helpx  "Frankfurt" -AND- co us)
   (st fl "Ft. Lauderdale" :dx2 :dx-adj "Ft Lauderdale" :dx-adj "Fort
Lauderdale")
   (st tx "Ft. Worth" :dx2 :dx-adj "Ft Worth" :dx-adj "Fort Worth")
   (st fl "Gainesville" :dx1)
   (st tx "Galveston" :dx1)
   (co it "Genova" :dx1 :dx-helpx "Genoa" -NOT- co us :invisible)
   (co uk "Glasgow" :dx1)
   (co de "Gottingen" :dx1 :invisible)
   (st nd "Grand Forks" :dx2)
   (st nc "Greenville$nc" :dx-helpx "Greenville" -AND- st nc)
   (co il "Haifa" :dx1 :invisible)
   (co de "Hamburg" :dx1 :invisible)
   (st nh "Hanover/Lebanon$nh"
       :dx-helpx "Lebanon" -AND- st nh
       :dx-helpx "Hanover" -AND- st nh)
   (co fi "Helsinki" :dx1 :invisible)
   (st pa "Hershey$pa" :dx-helpx "Hershey" -AND- st pa)
   (st hi "Honolulu" :dx1)
   (st tx "Houston" :dx-helpx "Houston" -NOT- ci "Fort Sam Houston")
   (co in "Hyderabad" :dx1 :invisible)
   (st in "Indianapolis" :dx1)
   (st ia "Iowa City" :dx2)
   ;;;;;;  Irvine -- see Los Angeles
```

```
    (co tr "Istanbul" :dx1 :invisible)
    (st ny "Ithaca" :dx1)
    (st ms "Jackson" :dx-helpx "Jackson" -AND- st ms) ; Jackson Hospital?
    (st fl "Jacksonville" :dx1)
    (st ma "Jamaica Plain" :dx2)
    (co il "Jerusalem" :dx1 :invisible)
    (st mi "Kalamazoo" :dx1)
    (co jp "Kanagawa" :sus "Kanagawa" :invisible)
    (st ks "Kansas City" :dx2)
    (co sd "Khartoum" :dx1 :invisible)
    (co jp "Kobe" :dx1 :invisible)
    (st qc "Laval" :dx-helpx "Laval" -AND- co ca)
    (co uk "Leeds" :dx1)
    (st ky "Lexington$ky" :dx-helpx "Lexington" -AND- st ky)
    (co pt "Lisboa" :dx1 :dx-helpx "Lisbon" -NOT- co us :invisible)
    (st ar "Little Rock" :dx2)
    (co uk "Liverpool" :dx1)
    (st on "London$on" :dx-helpx "London" -AND- co ca :dx-adj "London,
Ontario")
    (co uk "London$en" :dx-helpx "London" -AND- co uk :dx-adj "London,
England")
    (st ca "Los Angeles" :dx2
        :dx-helpx "Irvine" -AND- st ca)
    (st ky "Louisville" :dx1)
    (st tx "Lubbock" :dx1)
    (co nl "Maastricht" :dx1 :invisible)
    (st wi "Madison" :dx1)
    (co in "Madras" :dx1 :invisible)
    (co es "Madrid" :dx1 :invisible)
    (co uk "Manchester$uk" :dx-helpx "Manchester" -AND- co uk :dx-adj
"Manchester, England")
    (co de "Marburg" :sus "Marburg" :invisible)
    (co au "Melbourne" :dx-helpx "Melbourne" -NOT- co us :invisible) ; Florida
    (st tn "Memphis" :dx1) ; ignore Egypt
    (co mx "Mexico City" :dx2 :dx-all "Mexico DF" :dx-all "Mexico D.F." :dx-
all "Mexico D.F.,"  :invisible)
    (st fl "Miami" :dx-helpx "Miami" -AND- st fl :dx-adj "Miami Beach")
    (co uk "Middlesex$uk" :dx-helpx "Middlesex" -AND- co uk)
    (co it "Milan" :dx1 :dx "Milano" :invisible)
    (st wi "Milwaukee" :dx1)
    (st ny "Mineola" :dx-helpx "Mineola" -AND- st ny) ; just seems safer
    (st mn "Minneapolis" :dx1 :dx-helpx "Hennepin County" -AND- st mn)
    (co fr "Montpellier" :dx1 :invisible)
    (st qc "Montreal" :dx1)
    (st wv "Morgantown" :dx-helpx "Morgantown" -AND- st wv)
    (co ru "Moscow" :dx-helpx "Moscow" -NOT- co us :invisible)
    (co de "Munich" :dx1 :dx "Munchen" :invisible)
    (co de "Munster" :dx-helpx "Munster" -NOT- co us :invisible)
    (st tn "Nashville" :dx1)
    (co in "New Delhi" :dx2 :invisible)
    (st ct "New Haven" :dx2)
    (st la "New Orleans" :dx2)
    (st ny "New York City" :sus-adj "New York" :dx-adj "New York, NY"
        :dx-adj "New York, New York" :dx-adj "NY, NY"
        :dx "Brooklyn" :dx "Bronx" :dx-helpx "Harlem" -AND- co us)
    (st nj "Newark$nj" :dx-helpx "Newark" -NOT- st de)
    (co uk "Newcastle upon Tyne" :dx2 :dx-helpx "Newcastle" -AND- co uk)
    (st va "Norfolk$va" :dx-helpx "Norfolk" -AND- co us :dx-helpx "Portsmouth"
-AND- st va)
    (st ca "Oakland" :dx1)
    (st ok "Oklahoma City" :dx2)
```

```
    (st ne "Omaha" :dx1)
    (co jp "Osaka" :dx1 :invisible)
    (co no "Oslo" :dx1 :invisible)
    (co uk "Oxford$en" :dx-helpx "Oxford" -AND- co uk)
    (st oh "Oxford$oh" :dx-helpx "Oxford" -AND- st oh)
    (co it "Padua" :dx1 :invisible)
    (st ca "Palo Alto" :dx2 :dx-helpx "Stanford" -AND- st ca)
    (co fr "Paris" :dx1 :invisible)
    (co au "Perth" :dx1 :invisible)
    (st pa "Philadelphia" :dx1)
    (st az "Phoenix" :dx1 :dx-helpx "Scottsdale" -AND- st az)
    (st pa "Pittsburgh" :dx1) ; ignore calif.
    ;; Portsmouth, VA -- see Norfolk
    (co cz "Prague" :dx1 :invisible)
    (st nj "Princeton" :dx1)
    (st ri "Providence" :dx-helpx "Providence" -AND- st ri) ; Sisters of
Providence
    (st nc "Research Triangle" :dx2
      :dx-adj "Chapel Hill"
      :dx-helpx "Raleigh" -AND- st nc
      :dx-helpx "Durham" -AND- st nc
      :dx-all "Duke Durham")
    (st va "Richmond" :dx1 :dx-helpx "Richmond" -AND- co us)
    (st mn "Rochester$mn" :dx-helpx "Rochester" -AND- st mn :dx-all "Rochester
MN" :dx-all "Rochester Minnesota")
    (st ny "Rochester$ny" :dx-helpx "Rochester" -AND- st ny :dx-all "Rochester
NY" :dx-all "Rochester New York")
    (st ny "Rome$ny" :dx-helpx "Rome" -AND- st ny :dx-all "Rome NY")
    (co it "Rome$it" :dx-helpx "Rome" -NOT- co us :dx "Roma" :invisible)
    (co nl "Rotterdam" :dx1 :invisible)
    (st ca "Sacramento" :dx1
        :dx-helpx "Davis" -AND- st ca)
    (st ut "Salt Lake City" :dx2)
    (st tx "San Antonio" :dx2)
    (st ca "San Diego" :dx2 :dx-adj "La Jolla")
    (st ca "San Francisco" :dx2)
    (st ca "San Jose" :dx-helpx "San Jose" -AND- co us)
    (co br "Sao Paulo" :dx2 :invisible)
    (co jp "Sapporo" :dx1 :invisible)
    (st sk "Saskatoon" :dx1)
    (st ny "Schenectady" :dx1) ; General Electric Corporate Research
    (st wa "Seattle" :dx1)
    (co jp "Sendai" :dx1 :invisible)
    (st la "Shreveport" :dx1)
    (co uk "Southampton$uk" :dx-helpx "Southampton" -AND- co uk)
    (st mo "St. Louis" :dx2 :dx-adj "St Louis") ; French hospitals?
    (co se "Stockholm" :dx1 :invisible)
    (st ny "Stony Brook$ny" :dx-helpx "Stony Brook" -AND- st ny :dx-all "Stony
Brook NY")
    (co uk "Surrey" :dx1)
    (co au "Sydney" :dx-helpx "Sydney" -NOT- co us :invisible)
    (co tw "Taipei" :dx1 :invisible)
    (st fl "Tampa" :dx1)
    (co ge "Tbilisi" :dx1 :dx-all "T'bilisi" :invisible)
    (co il "Tel Aviv" :dx2 :dx "Tel-Aviv" :invisible)
    (co il "Tel Hashomer" :dx2 :invisible)
    (co jp "Tokyo" :dx1 :dx "Tokio" :invisible)
    (st oh "Toledo" :dx-helpx "Toledo" -AND- co us) ; Spain
    (st on "Toronto" :dx1)
    (st az "Tucson" :dx1)
    (co de "Ulm" :dx-all "Ulm Germany" :invisible)
```

```
   (st pa "University Park$pa" :dx-helpx "University Park" -AND- st pa) ;
Penn State
   (st il "Urbana" :dx1)
   (st bc "Vancouver" :dx1)
   (co pl "Warsaw" :dx-helpx "Warsaw" -NOT- co us :invisible)
   (st dc "Washington$dc"     :dx-helpx "Washington" -AND- st dc :dx-all
"Washington DC")
   (st vt "White River Junction" :dx2)
   (st de "Wilmington$de" :dx-helpx "Wilmington" -AND- st de :dx-all
"Wilmington Del.")
   (st nc "Wilmington$nc" :dx-helpx "Wilmington" -AND- st nc)
   (st mb "Winnipeg" :dx1)
   (st nc "Winston-Salem" :dx-adj "Winston-Salem" :dx-adj "Winston Salem")
   (co jp "Yonago" :dx1 :invisible)
   (co uk "York$uk" :dx-helpx "York" -AND- co uk)
   (co ch "Zurich" :dx1 :invisible)
   ))


(map nil #'make-an-institution
 '(
   ("Albany Medical College" ci "Albany"
    :dx2
    :dx-adj "Albany Medical Center"
    :edu-domain "amc")
   ("Albert Einstein School of Medicine" ci "New York City"
    :dx2
    :dx-adj "Albert Einstein College of Medicine"
    :dx-all "Albert Einstein Yeshiva"
    :dx-all "Albert Einstein New York"
    :domain "aecom.yu.edu")
   ("Albert Einstein Medical Center" ci "Philadelphia"
    :dx-helpx "Albert Einstein" -AND- ci "Philadelphia")
   ("Alfred I. duPont Institute" ci "Wilmington$de"
    :dx-all "Alfred I duPont Institute")
   ("American Heart Association" ci "Dallas"
    :dx2)
   ("Baylor College of Medicine" ci "Houston"
    :dx-adj "Texas Children's Hospital"
    :dx2)
   ("Beth Israel Deaconess Medical Center" ci "Boston"
    :dx2)
   ("Boston University" ci "Boston"
    :dx2
    :dx-adj "Boston City Hospital")
   ("Brown University" ci "Providence"
    :dx2
    :dx-adj "Rhode Island Hospital")
   ("Case Western Reserve University" ci "Cleveland"
    :dx2
    :dx-helpx "Case Western" -AND- st oh)
   ("Centers for Disease Control" ci "Atlanta"
    :dx2
    :dx-helpx "CDC" -AND- ci "Atlanta"
    :domain "cdc.gov")
   ("Children's National Medical Center" ci "Washington$dc"
    :dx2)
   ("Cleveland Clinic" ci "Cleveland"
    :dx-helpx "Cleveland Clinic" -AND- st oh)
   ("Cleveland Clinic Ft. Lauderdale" ci "Ft. Lauderdale"
```

```
     :dx-helpx "Cleveland Clinic" -AND- st fl
     :dx-all "Cleveland Clinic Florida"
     :dx-all "Cleveland Clinic Lauderdale")
    ("Columbia University" ci "New York City"
     :dx "Columbia-Presbyterian"
     :dx2
     :dx-helpx "Columbia" -AND- st ny
     :dx-all "Columbia Physicians Surgeons"
     :dx-helpx "College of Physicians and Surgeons" -AND- co US
     :edu-domain "columbia")
    ("Cornell University" ci "Ithaca"
     :dx-helpx "Cornell" -AND- ci "Ithaca"
     :edu-domain "cornell")
    ("Creighton University" ci "Omaha"
     :dx2)
    ("Dartmouth College" ci "Hanover/Lebanon$nh"
     :dx2)
    ("Dartmouth Medical School" ci "Hanover/Lebanon$nh"
     :dx2
     :dx-all "White River Junction VA"
     :dx-all "White River Junction Veterans"
     :ispartof "Dartmouth College")
    ("Duke University" ci "Research Triangle"
     :dx2)
    ("East Carolina University" ci "Greenville$nc"
     :dx2)
    ("Eastern Virginia Medical School" ci "Norfolk$va"
     :dx2)
    ("Emory University" ci "Atlanta"
     :dx2
     :edu-domain "emory")
    ("George Washington University" ci "Washington$dc"
     :dx2
     :dx-adj "George Washington Univ"
     :dx-adj "Geo. Washington University"
     :dx-adj "Geo Washington University"
     :dx-adj "Geo. Washington Univ")
    ("Georgia State University" ci "Atlanta"
     :dx2)
    ("Harvard University" ci "Boston"
     :dx "Harvard"
     :dx-adj "Massachusetts General"
     :dx-adj "Channing Laboratory"
     :dx-adj "Brigham and Women's Hospital"
     :dx-helpx "Children's Hospital" -AND- ci "Boston"
     :dx-helpx "Dana Farber" -AND- st ma
     :dx-all "Dana Farber Cancer"
     :dx-adj "Massachusetts Eye and Ear Infirmary"
     :dx-helpx "Eye and Ear Infirmary" -AND- st ma
     :edu-domain "harvard")
    ("Henry Ford Hospital" ci "Detroit"
     :dx2
     :www "http://www.henryfordhealth.org")
    ("Indiana University School of Medicine" ci "Indianapolis" ; cf. Indiana
U.
     :dx2)
    ("Johns Hopkins University" ci "Baltimore"
     :dx-adj "Johns Hopkins"
     :dx-helpx "John Hopkins" -AND- ci "Baltimore"
     :edu-domain "jhmi"
     :edu-domain "jhu")
```

```
("Louisiana State University" ci "Shreveport"
 :dx-adj "Louisiana State University Health"
 :dx-adj "Louisiana State University Medical"
 :dx-all "Lousiana State Shreveport"
 :edu-domain "lsu")
("Lovelace Medical Center" ci "Albuquerque"
 :dx2)
("Loyola University Stritch School of Medicine" ci "Chicago"
 :dx-helpx "Loyola" -AND- st il
 :dx-adj "Stritch School of Medicine"
 :zip "60153"
 :edu-domain "luc")
("Madigan Army Medical Center" ci "Fort Lewis"
 :dx2)
("Mayo Clinic Jacksonville" ci "Jacksonville"
 :dx-all "Mayo Clinic Jacksonville"
 :dx-helpx "Mayo" -AND- st fl)
("Mayo Clinic Rochester" ci "Rochester$mn"
 :dx-all "Mayo Clinic Rochester"
 :dx-helpx "Mayo" -AND- st mn)
("Mayo Clinic Scottsdale" ci "Phoenix"
 :dx-all "Mayo Clinic Scottsdale"
 :dx-helpx "Mayo" -AND- st az)

("Medical College of Georgia" ci "Augusta$ga"
 :dx2)
("Medical College of Ohio" ci "Toledo"
 :dx2)
("Medical College of Pennsylvania" ci "Philadelphia"
 :dx2
 :dx-helpx "MCP" -AND- ci "Philadelphia")
("Medical College of Virginia" ci "Richmond"
 :dx2
 :dx-helpx "MCV" -AND- ci "Richmond"
 :ispartof "Virginia Commonwealth University")
("Medical College of Wisconsin" ci "Milwaukee"
 :dx2
 :dx-helpx "MCW" -AND- ci "Milwaukee")

("Medical University of South Carolina" ci "Charleston"
 :dx2
 :zip "29425"
 :edu-domain "musc")
("Mercy Hospital of Pittsburgh" ci "Pittsburgh"
 :dx2
 :ispartof "Thomas Jefferson University"
 :domain "mercylink.org")
("Miami Children's Hospital" ci "Miami"
 :dx2)
("Miami University" ci "Oxford$oh"
 :dx-helpx "Miami University" -AND- st oh)
("Montefiore Medical Center" ci "New York City" ; Bronx
 :zip "10467"
 :dx2)
("Mount Sinai School of Medicine" ci "New York City"
 :dx2
 :dx-all "Mt Sinai School of Medicine"
 :dx-helpx "Mount Sinai Medical Center" -AND- st ny
 :dx-helpx "Mt Sinai Medical Center"    -AND- st ny
 :dx-helpx "Mt. Sinai Medical Center"   -AND- st ny)
("New England Medical Center" ci "Boston"
```

```
      :dx2)
     ("National Institutes of Health" ci "Bethesda"
      :dx2
      :dx-helpx "NIH" -AND- ci "Bethesda"
      :dx-adj "National Cancer Institute"
      :dx-all "National Heart Lung Blood Institute"
      :dx-all "National Institute of Neurological Disorders and Stroke"
      :dx-all "National Institute on Aging" ; Baltimore!!
      :dx-adj "National Institutes of Mental Health"
      :dx "NHLBI"
      :dx "NIDDK"
      :dx "NIMH"
      :dx "NINDS"
      :domain "nih.gov")
     ("Naval Medical Center Portsmouth" ci "Norfolk$va"
      :dx2
      :dx-helpx "Naval Medical Center Portsmouth" -AND- co us)
     ("New York University" ci "New York City"
      :dx2
      :edu-domain "nyu"
      :zip "10016")
     ("Northeastern University" ci "Boston"
      :dx-helpx "Northeastern University" -AND- ci "Boston")
     ("Northeastern Ohio Universities College of Medicine" ci "Akron"
      :dx2
      :edu-domain "neoucom")
     ("Northwestern University" ci "Chicago"
      :zip "60611"
      :dx2
      :edu-domain "nwu")
     ("Pennsylvania State University" ci "University Park$pa"
      :dx-helpx "Pennsylvania State University" -NOT- in "Pennsylvania State
University College of Medicine")
     ("Pennsylvania State University College of Medicine" ci "Hershey$pa"
      :dx2
      :dx-all "Penn State University College of Medicine"
      :dx-all "Penn State Hershey"
      :dx-all "Pennsylvania State Hershey"
      :dx-adj "Hershey Medical Center")
     ("Pharmacia and Upjohn" ci "Kalamazoo"
      :dx-all "Pharmacia Upjohn"
      :domain "pnu.com")
     ("Rockefeller University" ci "New York City"
      :dx2)
     ("Rush Presbyterian St. Luke's" ci "Chicago"
      :dx-adj "Rush Presbyterian St"
      :dx-adj "Rush-Presbyterian-St"
      :dx-adj "Rush Presbyterian St."
      :dx-adj "Rush-Presbyterian-St."
      :dx-adj "Rush Presbyterian Saint"
      :dx-adj "Rush-Presbyterian-Saint"
      :dx-helpx "Rush Presbyterian" -AND- ci "Chicago"
      :dx-helpx "Rush-Presbyterian" -AND- ci "Chicago"
      :dx-adj "Rush Children's Hospital"
      :dx-adj "Rush Medical College")
     ("Rutgers University" ci "Newark$nj"
      :dx2
      :dx-adj "New Jersey Medical School"
      :dx-all "University Medicine Dentistry New Jersey"
      :dx-helpx "UMDNJ" -AND- co us
      :dx-adj "Robert Wood Johnson Medical School")
```

```
    ("Saint Louis University" ci "St. Louis"
     :dx2)
    ("Scottish Rite Children's Medical Center" ci "Atlanta"
     :dx2
     :dx-helpx "Scottish Rite Children's" -AND- ci "Atlanta"
     :domain "choa.org")
    ("Stanford University" ci "Palo Alto"
     :dx2
     :www "http://www.stanford.edu"
     :edu-domain "stanford")
    ("SUNY Brooklyn" ci "New York City"
     :dx-all "State University New York Brooklyn"
     :dx-all "Health Science Center Brooklyn"
     :edu-domain "hscbklyn")
    ("SUNY Buffalo" ci "Buffalo"
     :dx-all "State University New York Buffalo")
    ("SUNY Stony Brook" ci "Stony Brook$ny"
     :dx-all "SUNY Stony Brook"
     :dx-all "State University New York Stony Brook"
     :edu-domain "sunysb")
    ("Temple University" ci "Philadelphia"
     :dx2)
    ("Texas Tech University" ci "Lubbock"
     :dx2)
    ("Thomas Jefferson University" ci "Philadelphia"
     :dx2
     :dx-adj "Jefferson Medical College")
    ("Tufts University" ci "Boston"
     :dx2
     :dx-helpx "Tufts" -AND- ci "Boston"
     :edu-domain "tufts")
    ("Tulane University" ci "New Orleans"
     :dx2
     :dx-adj "Tulane Hospital for Children"
     :dx-helpx "Tulane" -AND- st la ;  no other campuses I hope
     :edu-domain "tulane")
    ("Uniformed Services University" ci "Bethesda"
     :dx2
     :domain "usuhs.mil")

    ("University of Alabama" ci "Birmingham$al"
     :dx2)
    ("University of Arizona" ci "Tucson"
     :dx2
     :edu-domain "arizona")
    ("University of Arkansas" ci "Little Rock"
     :dx2)
    ("University of California Berkeley" ci "Berkeley"
     :dx-all "University of California Berkeley"
     :edu-domain "berkeley")
    ("University of California Davis" ci "Sacramento"
     :dx-all "University of California Davis")
    ("University of California Irvine" ci "Los Angeles" ; Irvine is in LA+/-
     :dx-all "University of California Irvine")
    ("University of California Los Angeles" ci "Los Angeles"
     :dx-adj "University of California, Los Angeles" ; U. of So. Cal., LA if
dx-all
     :dx "UCLA"
     :dx "Harbor-UCLA"
     :edu-domain "ucla")
    ("University of California San Diego" ci "San Diego"
```

```
 :dx-all "University of California San Diego"
 :dx "UCSD"
 :edu-domain "ucsd")
("University of California San Francisco" ci "San Francisco"
 :dx "UCSF"
 :dx-all "University of California San Francisco"
 :dx-adj "California Pacific Medical Center"
 :edu-domain "ucsf")
("University of Chicago" ci "Chicago"
 :dx2
 :zip "60637")
("University of Cincinnati" ci "Cincinnati"
 :dx2
 :edu-domain "uc")
("University of Colorado Health Sciences Center" ci "Denver"
 :dx2
 :dx-helpx "University of Colorado" -AND- ci "Denver"
 :dx-helpx "National Jewish" -AND- ci "Denver"
 :dx-adj "National Jewish Medical and Research Center"
 :domain "njc.org")
("University of Connecticut" ci "Farmington$ct"
 :dx2)
("University of Florida" ci "Gainesville" ; cf U. South Florida
 :dx2)
("University of Hawaii" ci "Honolulu"
 :dx2)
("University of Illinois Chicago" ci "Chicago"
 :dx-all "University of Illinois Chicago")
("University of Iowa" ci "Iowa City"
 :dx2)
("University of Kansas" ci "Kansas City"
 :dx2)
("University of Kentucky" ci "Lexington$ky"
 :dx2)
("University of Louisville" ci "Louisville"
 :dx2)
("University of Maryland" ci "Baltimore"
 :dx2)
("University of Miami" ci "Miami"
 :dx2
 :sus-helpx "Miami University" -AND- st fl
 :dx-helpx "Jackson Memorial" -AND- ci "Miami"
 :dx-adj "Bascom Palmer"
 :edu-domain "miami")
("University of Michigan" ci "Ann Arbor"
 :dx2)
("University of Minnesota" ci "Minneapolis"
 :dx2
 :dx-adj "Hennepin County Medical Center"
 :zip "55455"
 :edu-domain "umn")
("University of Mississippi" ci "Jackson"
 :dx2
 :dx-helpx "UMC" -AND- ci "Jackson")
("University of Missouri" ci "Columbia$mo"
 :dx2)
("University of Nebraska" ci "Omaha"
 :dx2)
("University of New Mexico" ci "Albuquerque"
 :dx2)
("University of North Carolina" ci "Research Triangle"
```

```
 :dx2)
("University of North Dakota" ci "Grand Forks"
 :dx2)
("University of Oklahoma" ci "Oklahoma City"
 :dx2)
("University of Pennsylvania" ci "Philadelphia"
 :dx2
 :dx-adj "Children's Hospital of Philadelphia"
 :dx-all "Fox Chase Cancer"
 :dx-helpx "Fox Chase" -AND- ci "Philadelphia"
 :edu-domain "chop")
("University of Pittsburgh" ci "Pittsburgh"
 :dx2
 :dx-adj "Children's Hospital of Pittsburgh"
 :edu-domain "pitt")
("University of Rochester" ci "Rochester$ny"
 :dx2
 :dx-adj "Strong Memorial Hospital")
("University of South Florida" ci "Tampa"
 :dx2)
("University of Southern California" ci "Los Angeles"
 :dx2
 :dx-helpx "Children's Hospital" -AND- ci "Los Angeles"
 :edu-domain "usc")
("University of Tennessee" ci "Memphis"
 :dx2
 :dx-all "St Jude Children's Research Hospital Memphis")
("University of Texas Houston" ci "Houston"
 :dx2
 :dx-helpx "University of Texas" -AND- ci "Houston"
 :dx-helpx "M.D. Anderson" -AND- ci "Houston"
 :dx-all "M.D. Anderson Cancer Center"
 :dx-all "St Luke's Episcopal Hospital"
 :dx-all "St. Luke's Episcopal Hospital" :domain "sleh.com"
 :edu-domain "tmc")
("University of Texas Medical Branch" ci "Galveston"
 :dx2
 :edu-domain "utmb"
 :zip "77550")
("University of Texas San Antonio" ci "San Antonio"
 :dx-all "University of Texas San Antonio")
("University of Texas Southwestern" ci "Dallas"
 :dx2
 :dx-all "University of Texas Southwestern"
 :dx-helpx "University of Texas" -AND- ci "Dallas"
 :dx-all "Parkland Hospital"
 :zip "75235")
("University of Utah" ci "Salt Lake City"
 :dx2
 :edu-domain "utah")
("University of Virginia" ci "Charlottesville"
 :dx2)
("University of Washington" ci "Seattle"
 :dx2
 :dx-adj "Harborview Medical Center"
 :dx-all "Fred Hutchinson Cancer")
("University of Wisconsin" ci "Madison"
 :dx2
 :dx-adj "University of Wisconsin-Madison"
 :edu-domain "wisc")
```

```
 ("Vanderbilt University" ci "Nashville"
  :dx "Vanderbilt")
 ("Virginia Commonwealth University" ci "Richmond"
  :dx2)
 ("Virginia Mason Medical Center" ci "Seattle"
  :dx2
  :www "http://www.vmmc.org"
  :zip "98111")
 ("Wake Forest University" ci "Winston-Salem"
  :dx2
  :dx-helpx "Bowman Gray" -AND- st nc
  :edu-domain "wfu"
  :zip "27157")
 ("Walter Reed Army Medical Center" ci "Washington$dc"
  :dx-adj "Walter Reed")
 ("Washington University" ci "St. Louis"
  :dx-helpx "Washington University" -AND- st mo ; Geo. Wash. U.
  :zip "63110"
  :edu-domain "wustl")
 ("Wayne State University" ci "Detroit"
  :dx2
  :dx-adj "Harper Hospital")
 ("Weill Medical College" ci "New York City"
  :dx2
  :dx-adj "Cornell University Medical College" ; old name
  :dx-adj "Weil Medical College" ; common mis-spelling
  :www "http://www.med.cornell.edu"
  :domain "med.cornell.edu"
  :ispartof "Cornell University")
 ("West Virginia University" ci "Morgantown"
  :dx2)
 ("Winthrop University Hospital" ci "Mineola"
  :dx2
  :domain "winthrop.org")
 ("Yale University" ci "New Haven"
  :dx "Yale"
  :edu-domain "yale")

;; **************
;; *** Canada ***

("McGill University" ci "Montreal"
 :dx2
 :dx-adj "Montreal General Hospital"
 :domain "mcgill.ca")
("Universite of Laval" ci "Laval"
 :dx-helpx "Universite Laval" -AND- co ca
 :domain "ulaval.ca")
("University of Alberta" ci "Edmonton"
 :dx2)
("University of British Columbia" ci "Vancouver"
 :dx2
 :domain "ubc.ca")
("University of Calgary" ci "Calgary"
 :dx2
 :dx-helpx "Alberta Children's Hospital" -AND- co ca
 :domain "ucalgary.ca")
("University of Manitoba" ci "Winnipeg"
 :dx2
 :dx-helpx "St Boniface General Hospital" -AND- co ca
 :dx-helpx "St. Boniface General Hospital" -AND- co ca
```

```
   :domain "umanitoba.ca")
  ("University of Montreal" ci "Montreal"
   :dx2
   :dx-helpx "Cartierville Hospital" -AND- st qc
   :dx-adj "Universite de Montreal")
  ("University of Saskatchewan" ci "Saskatoon"
   :dx2)
  ("University of Toronto" ci "Toronto"
   :dx2
   :dx-helpx "Hospital for Sick Children" -AND- ci "Toronto"
   :domain "utoronto.ca")
  ("University of Western Ontario" ci "London$on"
   :dx2
   :domain "uwo.ca")

  ;; **********************
  ;; *** United Kingdom ***
  ;; See: http://www.londonmedicine.org.uk/medlist.htm

  ("Charing Cross and Westminster Medical School" ci "London$en"
   :dx2
   :dx-all "Charing Cross London"
   :ispartof "Imperial College School of Medicine")
  ("Eastman Dental Institute and Hospital" ci "London$en"
   :dx2
   :domain "eastman.ucl.ac.uk"
   :www "http://www.eastman.ucl.ac.uk"
   :ispartof "UCL Medical School")
  ("Great Ormond Steet Hospital for Children" ci "London$en"
   :dx-adj "Great Ormond Street Hospital"
   :ispartof "Institute of Child Health")
  ("GKT School of Medicine" ci "London$en"
   :dx-all "Guy's King's St Thomas' School of Medicine"
   :dx-adj "United Medical and Dental Schools"
   :www "http://www.kcl.ac.uk/depsta/medicine/index.html")
  ("Guy's Hospital" ci "London$en"
   :dx2
   :ispartof "GKT School of Medicine")

  ("Imperial College School of Medicine" ci "London$en"
   :dx2
   :sus-adj "Imperial College"
   :www "http://www.med.ic.ac.uk"
   :domain "med.ic.ac.uk")
  ("Imperial College School of Medicine at the National Heart and Lung
Institute" ci "London$en"
   :dx2
   :dx-helpx "National Heart and Lung Institute" -AND- co uk
   :ispartof "Imperial College School of Medicine")
  ("Imperial College School of Medicine at St. Mary's" ci "London$en"
   :dx2
   :dx-helpx "St Mary's"  -AND- ci "London$en"
   :dx-helpx "St. Mary's" -AND- ci "London$en"
   :ispartof "Imperial College School of Medicine")

  ("Institute of Cancer Research" ci "Surrey"
   :dx-helpx "Institute of Cancer Research" -AND- co uk
   :www "http://www.icr.ac.uk")
  ("Institute of Child Health" ci "London$en"
   :dx-helpx "Institute of Child Health" -AND- co uk
   :domain "ich.ucl.ac.uk"
```

```
      :www "http://www.ich.ucl.ac.uk"
      :ispartof "UCL Medical School")
     ("Institute of Neurology" ci "London$en"
      :dx-helpx "Institute of Neurology" -AND- co uk
      :dx-all "Neurology Queen Square"
      :domain "ion.ucl.ac.uk"
      :www "http://www.ion.ucl.ac.uk"
      :ispartof "UCL Medical School")
     ("Institute of Ophthalmology" ci "London$en"
      :dx-helpx "Institute of Ophthalmology" -AND- co uk
      :www "http://www.ucl.ac.uk/ioo/"
      :ispartof "UCL Medical School")
     ("Institute of Orthopaedics" ci "Middlesex$uk"
      :dx-helpx "Institute of Orthopaedics" -AND- co uk
      :ispartof "UCL Medical School")
     ("Institute of Psychiatry" ci "London$en"
      :dx-helpx "Institute of Psychiatry" -AND- co uk
      :www "http://www.iop.bpmf.ac.uk")
     ("Institute of Urology and Nephrology" ci "London$en"
      :dx-helpx "Institute of Urology" -AND- co uk
      :dx-helpx "Institute of Nephrology" -AND- co uk
      :ispartof "UCL Medical School")

     ("King's College School of Medicine and Dentistry" ci "London$en"
      :dx2
      :ispartof "GKT School of Medicine")
     ("Middlesex Hospital" ci "London$en"
      :dx-helpx "Middlesex Hospital" -AND- co uk
      :ispartof "UCL Medical School")
     ("Leeds General Infirmary" ci "Leeds"
      :dx2)
     ("London School of Hygiene and Tropical Medicine" ci "London$en"
      :dx2
      :domain "lshtm.ac.uk"
      :ispartof "University of London")
     ("Royal Brompton Hospital and/or Trust" ci "London$en"
      :dx-adj "Royal Brompton")
     ("Royal Free and University College Medical School" ci "London$en"
      :dx-all "Royal Free London"
      :dx-helpx "Royal Free" -AND- co uk
      :domain "rfhsm.ac.uk"
      :domain "rfc.ucl.ac.uk"
      :www "http://www.rfhsm.ac.uk"
      :ispartof "UCL Medical School")
     ("Royal Postgraduate Medical School" ci "London$en"
      :dx2
      :ispartof "Imperial College School of Medicine")
     ("School of Pharmacy" ci "London$en"
      :dx-helpx "School of Pharmacy" -AND- ci "London$en"
      :www "http://www.ulsop.ac.uk"
      :domain "ulsop.ac.uk"
      :ispartof "University of London")
     ("St. Bartholomew's and the Royal London Hospital School of Medicine and
Dentistry" ci "London$en"
      :dx-helpx "St Bartholomew's" -AND- co uk
      :dx-helpx "St. Bartholomew's" -AND- co uk
      :dx-adj "Royal London School of Medicine"
      :www "http://www.mds.qmw.ac.uk/"
      :ispartof "University of London")
     ("St. George's Hospital Medical School" ci "London$en"
      :dx-all "St George's Hospital Medical School"
```

```
     :dx-helpx "St George's" -AND- co uk
     :dx-helpx "St. George's" -AND- co uk
     :domain "sghms.ac.uk"
     :www "http://www.sghms.ac.uk"
     :ispartof "University of London")
    ("St. Thomas' Hospital" ci "London$en"
     :dx2
     :dx-adj "St Thomas' Hospital"
     :ispartof "GKT School of Medicine")
    ("UCL Medical School" ci "London$en"
     :dx2
     :dx-adj "UCL School of Medicine"
     :ispartof "University College London")
    ("University College London" ci "London$en"
     :dx2
     :domain "ucl.ac.uk")
    ;;;   university of EAST LONDON
    ;;;   university of GREENWICH
    ;;;   university of HERTFORDSHIRE
    ;;;    ("University of North London" ci "London$en"
    ;;;     :dx2
    ;;;     :domain "unl.ac.uk"
    ;;;     :www "http://www.unl.ac.uk")
    ("University of Edinburgh" ci "Edinburgh"
     :dx2
     :dx-all "Royal Infirmary Edinburgh"
     :dx-all "Edinburgh Royal Infirmary")
    ("University of London" ci "London$en"
     :dx2
     :domain "lon.ac.uk"
     :www "http://www.lon.ac.uk")
    ("University of Manchester" ci "Manchester$uk"
     :dx2)
    ("University of Oxford" ci "Oxford$en"
     :dx-adj "Oxford University"
     :dx-adj "John Radcliffe Hospital"
     :dx-adj "Oxford Radcliffe Hospital"
     :dx-helpx "Radcliffe" -AND- ci "Oxford$uk"
     :dx-helpx "Osler Chest Unit" -AND- co uk
     :dx-helpx "Churchill Hospital Oxford" -AND- co uk
     :dx2
     :domain "ox.ac.uk")
    ("University of Southampton" ci "Southampton$uk"
     :dx2
     :dx-adj "Southampton University"
     :dx-adj "Southampton General Hospital"
     :domain "soton.ac.uk")
    ("University of Westminster" ci "London$en"
     :dx2
     :domain "wmin.ac.uk"
     :www "http://www.wmin.ac.uk")
    ("University of York" ci "York$uk"
     :dx2)

   ))

   ;; *************************
   ;; *** Invisible Foreign ***

(map nil #'make-an-invisible-institution
 '(
```

```
    (co lb :dx-adj "American University of Beirut") ; has NYC P.O. address!!!!
    (co il :dx-adj "Hadassah University")
    (co se :dx-adj "Karolinska Institute" :dx-adj "Karolinska Hospital")
    (co il :dx     "Technion")
    (co be :dx-adj "Universite Catholique de Louvain" :dx "Louvain")
    (co jo :dx-adj "University of Jordan")
    ))

'done

@@file countries

(prepare-to-load-places)
(prepare-to-load-rules)

(make-a-planet)

(map nil #'make-a-country
  ;; TLDs from: http://www.norid.no/domreg.html
 '(
    (us "USA"         :tld gov :tld edu :tld mil :dx-adj "USA."
        :place-var *place-usa*)

    (uk "United Kingdom" :dx2 :dx-adj "Great Britain" :dx-adj "UK." :tld gb
        :sus "England" :dx-helpx "Staffordshire" -AND- co uk
        :dx "Scotland"
        :dx-adj "Northern Ireland"
        :dx-helpx "Wales" -NOT- 2 (co au co hk co cn) ; New South Wales;
Prince of Wales Hosp. in Hong Kong
        :dx "Guernsey" :tld gg
        :dx-adj "Isle of Man" :tld im
        :sus-helpx "Jersey" -NOT- st nj :tld je
        :place-var *place-uk*)

    (ar "Argentina"   :dx1)
    (au "Australia"   :dx1 :dx-adj "New South Wales" :dx "Queensland")
    (at "Austria"     :dx1)
    (bd "Bangladesh"  :dx1)
    (be "Belgium"     :dx1)
    (bo "Bolivia"     :dx1)
    (br "Brazil"      :dx1)
    (bg "Bulgaria"    :dx1)
    (ca "Canada"      :dx1 :place-var *place-canada*)
    (cl "Chile"       :dx1)
    (cn "China"       :dx-adj "Peoples Republic of China" :dx-adj "People's
Republic of China" :dx-adj "P.R. China")
    (co "Colombia"    :dx1c) ; Clumbia mis-spellings prob. more likely
    (cr "Costa Rica"  :dx2)
    (cu "Cuba"        :dx1c)
    (cy "Cyprus"      :dx1)
    (cz "Czech Republic" :dx2 :dx "Czech" :dx "Czechoslovakia" :tld cs)
    (dk "Denmark"     :dx1)
    (ec "Ecuador"     :dx1)
    (eg "Egypt"       :dx1)
    (ee "Estonia"     :dx1)
    (et "Ethiopia"    :dx1)
    (fi "Finland"     :dx1)
    (fr "France"      :dx1)
    (ge "Georgia"     :sus-helpx "Georgia" -NOT- co us :dx-helpx "Georgia" -
AND- co ge)
    (de "Germany"     :dx1)
```

```
    (gr "Greece"      :dx1)
    (gt "Guatemala"   :dx1)
    (ht "Haiti"       :dx1)
    (hn "Honduras"    :dx1)
    (hk "Hong Kong"   :dx2)
    (hu "Hungary"     :dx1)
    (is "Iceland"     :dx1)
    (in "India"       :dx1c)
    (id "Indonesia"   :dx1)
    (ie "Ireland"     :sus-helpx "Ireland" -NOT- st ni :place-var *place-
ireland*)
    (il "Israel"      :dx1c) ; "Israel Morgenstern Ctr for.."?
    (it "Italy"       :dx1 :dx "Italia")
    (jm "Jamaica"     :dx1c :sus-helpx "Jamaica" -NOT- ci "Jamaica Plain")
    (jp "Japan"       :dx1)
    (jo "Jordan"      :sus-helpx "Jordan" -NOT- co wg)
    (ke "Kenya"       :dx1)
    (kr "Korea"       :dx1 :tld kp)
    (kw "Kuwait"      :dx1)
    (lb "Lebanon"     :dx1c) ; Cedars of Lebanon; Lebanon, NH
    (ml "Mali"        :dx1)
    (my "Malaysia"    :dx1)
    (mx "Mexico"      :sus-helpx "Mexico" -NOT- st nm) ; may be better as NOT
co us
    (nl "Netherlands" :dx1)
    (nz "New Zealand" :dx2) ; Zealand in Denmark
    (ni "Nicaragua"   :dx1)
    (no "Norway"      :dx1)
    (pk "Pakistan"    :dx1)
    (pa "Panama"      :dx1c) ; Panama City, FL
    (pe "Peru"        :dx1c) ; Peru, IN
    (ph "Philippines" :dx1)
    (pl "Poland"      :dx1)
    (pt "Portugal"    :dx1)
    (ro "Romania"     :dx1)
    (ru "Russia"      :dx1 :tld su :sus "USSR" :sus "Soviet")
    (sa "Saudi Arabia" :dx2)
    (sg "Singapore"   :dx1)
    (za "South Africa" :dx2 :dx-adj "Orange Free State")
    (es "Spain"       :dx1)
    (se "Sweden"      :dx1) ; Sweden Hospital?
    (ch "Switzerland" :dx1)
    (sy "Syria"       :dx1)
    (tw "Taiwan"      :dx1 :dx-adj "ROC." :dx-helpx "Republic of China" -NOT-
co cn)
    (tz "Tanzania"    :dx1)
    (th "Thailand"    :dx1)
    (tr "Turkey"      :dx1)
    (ug "Uganda"      :dx1)
    (ua "Ukraine"     :dx1)
    (ae "United Arab Emirates" :dx2)
    (ve "Venezuela"   :dx1)
    (vn "Viet Nam"    :dx2)
    (yu "Yugoslavia"  :dx1 :dx "Serbia" :dx "Montenegro")
    (zm "Zambia"      :dx1)
    (zr "Zaire"       :dx1)
    (zw "Zimbabwe"    :dx1)

    (ad "Andorra" :dx1)
    (af "Afghanistan" :dx1)
    (al "Albania" :dx1)
```

```
    (am "Armenia" :dx1)
    (ao "Angola" :dx1)
    (az "Azerbaijan" :dx1)
    (ba "Bosnia and Herzegowina" :dx "Bosnia" :dx "Herzegowina" :dx
"Herzegovina")
    (bf "Burkina Faso" :dx2)
    (bh "Bahrain" :dx1)
    (bi "Burundi" :dx1)
    (bj "Benin" :dx1)
    (bn "Brunei Darussalam" :dx "Brunei")
    (bt "Bhutan" :dx1)
    (bw "Botswana" :dx1)
    (by "Belarus" :dx1)
    (bz "Belize" :dx1)
    (cf "Central African Republic" :dx2)
    (cg "Congo" :dx1)
    (ci "Ivory Coast" :dx2 :dx-adj "Cote d'Ivoire")
    (cm "Cameroon" :dx1)
    (dj "Djibouti" :dx1)
    (do "Dominican Republic" :dx2)
    (dz "Algeria" :dx1)
    (eh "Western Sahara" :dx2)
    (er "Eritrea" :dx1)
    (ga "Gabon" :dx1)
    (gf "French Guiana" :dx2)
    (gh "Ghana" :dx1)
    (gi "Gibraltar" :dx1)
    (gl "Greenland" :dx1)
    (gm "Gambia" :dx1)
    (gn "Guinea" :dx-helpx "Guinea" -NOT- 2 (co gw co gq))
    (gp "Guadeloupe" :dx1)
    (gq "Equatorial Guinea" :dx2)
    (gw "Guinea-Bissau" :dx1 :dx-adj "Guinea Bissau")
    (gy "Guyana" :dx1)
    (hr "Croatia" :dx1)
    (iq "Iraq" :dx1)
    (ir "Iran" :dx1)
    (kg "Kyrgystan" :dx1)
    (kh "Cambodia" :dx1)
    (kz "Kazakhstan" :dx1)
    (la "Laos" :dx1 :dx-all "Lao Republic")
    (li "Liechtenstein" :dx1)
    (lk "Sri Lanka" :dx2)
    (lr "Liberia" :dx1)
    (ls "Lesotho" :dx1)
    (lt "Lithuania" :dx1)
    (lu "Luxembourg" :dx1)
    (lv "Latvia" :dx1)
    (ly "Libya" :dx1)
    (ma "Morocco" :dx1)
    (mc "Monaco" :dx1)
    (md "Moldova" :dx1)
    (mg "Madagascar" :dx1)
    (mk "Macedonia" :dx1)
    (mm "Myanmar" :dx1)
    (mn "Mongolia" :dx1)
    (mo "Macau" :dx1)
    (mt "Malta" :dx1)
    (mw "Malawi" :dx1)
    (mr "Mauritania" :dx1)
    (mz "Mozambique" :dx1)
```

```
(na "Namibia" :dx1)
(ne "Niger" :dx1)
(ng "Nigeria" :dx1)
(np "Nepal" :dx1)
(om "Oman" :dx1)
(ps "Palestine" :dx1)
(py "Paraguay" :dx1)
(qa "Qatar" :dx1)
(rw "Rwanda" :dx1)
(sd "Sudan" :dx1)
(si "Slovenia" :dx1)
(sk "Slovakia" :dx1 :dx-adj "Slovak Republic")
(sl "Sierra Leone" :dx2)
(sm "San Marino" :dx2)
(sn "Senegal" :dx1)
(so "Somalia" :dx1)
(sr "Surinam" :dx1)
(sv "El Salvador" :dx2)
(sz "Swaziland" :dx1)
(td "Chad" :dx1)
(tg "Togo" :dx1)
(tj "Tajikistan" :dx1)
(tm "Turkmenistan" :dx1)
(tn "Tunisia" :dx1)
(tp "East Timor" :dx2)
(uy "Uruguay" :dx1)
(uz "Uzbekistan" :dx1)
(va "Vatican City" :dx2 :dx-adj "Holy See")
(wg "Western Jordan and Gaza" :dx-adj "Western Jordan" :dx "Gaza")
(ye "Yemen" :dx1)

(ac "Atlantic islands" :dx-island "Ascension"
 :tld bm :dx "Bermuda"
 :tld bv :dx-island "Bouvet"
 :tld cv :dx-adj "Cape Verde"
 :tld fk :dx-islands "Falkland" :dx "Falklands"
 :tld fo :dx-islands "Faroe"
 :tld st :dx-all "Sao Tome Principe"
 :tld sh :dx-adj "St. Helena" :dx-adj "St Helena"
 :tld pm :dx-all "St Pierre Miquelon"
 :tld gs :dx-islands "South Georgia" :dx-islands "South Sandwich"
 )
(as "Pacific islands" :dx-adj "American Samoa"
 :tld cx :dx-island "Christmas"
 :tld ck :dx-islands "Cook"
 :tld fj :dx "Fiji"
 :tld pf :dx-adj "French Polynesia"
 :tld gu :dx "Guam"
 :tld ki :dx "Kiribati"
 :tld mh :dx-islands "Marshall"
 :tld fm :dx "Micronesia"
 :tld nr :dx "Nauru"
 :tld nc :dx-adj "New Caledonia"
 :tld nu :dx "Niue"
 :tld nf :dx-island "Norfolk"
 :tld mp :dx-islands "Northern Mariana"
 :tld pw :dx "Palau"
 :tld pg :dx-adj "Papua New Guinea"
 :tld pn :dx "Pitcairn"
 :tld ws :dx "Samoa"
 :tld sb :dx-islands "Solomon"
```

```
      :tld tk :dx "Tokelau"
      :dx "Tonga"  ; :tld to is being sold out
      :dx "Tuvalu" ; :tld tv is being sold out
      :tld um  ; United States Minor Outlying Islands
      :tld vu :dx "Vanuatu"
      :tld wf :dx-islands "Wallis and Futuna"
      )
     (aw "Caribbean islands" :dx2 :dx "Aruba"
      :tld ai :dx "Anguilla"
      :tld ag :dx "Antigua" :dx "Barbuda"
      :tld bs :dx "Bahamas"
      :tld bb :dx "Barbados"
      :tld ky :dx-islands "Cayman"
      :tld dm :dx "Dominica"
      :tld gd :dx "Grenada"
      :tld mq :dx "Martinique"
      :tld ms :dx "Montserrat"
      :tld an :dx-adj "Netherlands Antilles"
      :tld pr :dx-adj "Puerto Rico"
      :tld kn :dx-all "St Kitts" ; ignore Nevis
      :tld lc :dx-adj "Saint Lucia"
      :tld vc :dx-adj "Saint Vincent" :dx "Grenadines"
      :tld tt :dx-all "Trinidad Tobago"
      :tld tc :dx-all "Turks Caicos"
      :tld vg :dx-adj "Virgin Islands" ; British
      :tld vi ; American
      )
     (km "Indian Ocean islands" :dx "Comoros"
      :tld io :dx-adj "British Indian Ocean Territory"
      :dx-island "Cocos" :dx-island "Keeling" ; :tld cc for sale
      :tld tf :dx-adj "French Southern Territories"
      :tld hm :dx-island "Heard" :dx-island "McDonald"
      :tld mv :dx "Maldives"
      :tld mu :dx "Mauritius"
      :tld yt :dx "Mayotte"
      :tld re :sus "Reunion"
      :tld sc :dx "Seychelles"
      )
     (aq "Remote regions" :dx "Antarctica"
      :tld nt :dx-adj "Neutral Zone"
      :tld sj :dx-island "Svalbard" :dx-island "Jan Mayen"
      )
     ))

(boot::load-logical-path-file "source:states")

@@file hello.html

<html>
<head>
<title>Hello World</title>
</head>
<body>
<center><h3>Getting the Webserver to Say Hello</h3></center>
<p>
<form action="http://10.1.0.4/cgi-bin/1.cgi">
<input type="hidden" name="fn" value="say_hello">

<table>
<tr>
<td valign="top"><b>Say it:</b></td>
```

```
<td valign="top">
<input type="radio" name="repetition" value="1" CHECKED>Once
<br>
<input type="radio" name="repetition" value="2">Twice
<br>
<input type="radio" name="repetition" value="3">Three times
<br>
<input type="radio" name="repetition" value="4">Four times
</td>
</tr>
<tr><td> </td></tr>
<tr>
<td colspan="2" align="center"><input type="submit"></td>
</tr>
</table>

</form>
</body>
</html>

@@file lisp-macros

;;; Adapted from Norvig p. 337. (C) 1999-2000 by John Sotos. All Rights
Reserved.
;;; My additions: deconstructor-fn parameter
;;;               erase-<resource> method
;;;               allocate-mass-<resource> method
;;;               < print-alloc-p > and notfication of allocation
(defmacro defresource (name &key constructor
                            (initial-copies 0)
                            (size (max initial-copies 10))
                            (extension-size (max 10 (floor size 2)))
                            (destructor-fn #'identity))
  "< constructor > should be a form, eg (make-record :rect)
   < destructor-fn > should be a function of one argument, eg
   #'(lambda (x) (dispose-record x)).  This function is applied to all
   resources in the resource storage *up to the fill pointer*.  In other
   words, anything that has been allocated and remains allocated will not
   be clobbered."
  (flet ((symbol (&rest args)
           (declare (dynamic-extent args))
           "Concatenate symbols or strings to form an interned symbol. Norvig
p. 302"
           (intern (format nil "~{~a~}" args))))
    (let ((resource      (symbol name '-resource))
          (deallocate    (symbol 'deallocate- name))
          (allocate      (symbol 'allocate- name))
          (allocate-mass (symbol 'allocate-mass- name))
          (erase         (symbol 'erase- name))
         #-production-system
          (inspect       (symbol 'inspect- name))
         #-production-system
          (length        (symbol 'length- name))
          (extensionsize (gensym)))
      `(let ((,resource (make-array ,size :fill-pointer 0 :adjustable t))
             (,extensionsize ,extension-size)
             (print-alloc-p t))

         (defun ,allocate ()
           "Get an element from the resource pool, or make one."
           (if (= (fill-pointer ,resource) 0)
```

```
            (progn
              #-production-system
              (when print-alloc-p
                #+hera
                (format t "~%;;; Allocating ~S #~S." (quote ,name) (length
,resource))
                ; (princ ,(concatenate 'string #.(string #\Newline)
"Allocating " (symbol-name name)))
                )
              ,constructor)
            (vector-pop ,resource)))

        (defun ,deallocate (,name)
          "Place a no-longer-needed element back in the pool."
          (vector-push-extend ,name ,resource ,extensionsize))

        (defun ,erase ()
          "Destroys all unallocated resources without freeing the storage."
          ;; Use of < map > ensures that only the visible part of the
          ;; resource vector is processed.
          (map nil ,destructor-fn ,resource))

        #-production-system
        (defun ,inspect ()
          (inspect ,resource))

        #-production-system
        (defun ,length ()
          (length ,resource))

        (defun ,allocate-mass (n)
          ;; Ensures < n > unused pre-formed resources available in resource
stack.
          (let ((n-new (- n (fill-pointer ,resource))))
            (dotimes (i n-new)
              (,allocate))))

        ,(when (> initial-copies 0)
          `(,allocate-mass ,initial-copies))
        ',name))))

;;; ---------------------------------------------------------------------------
-----------|
;;; LISP MACROS

(defmacro dovec ((elt index vec) &body body)
   (let ((vecvar (gensym)))
      `(let (,elt
             (,vecvar ,vec))
         (dotimes (,index (length ,vecvar))
            (setq ,elt (aref ,vecvar ,index))
            ,@body))))

(defmacro let-de (args &body body)
  `(let ,args
     (declare ,(cons 'dynamic-extent (mapcar #'first args)))
     ,@body))

(defmacro let*de (args &body body)
  `(let* ,args
```

```
      (declare ,(cons 'dynamic-extent (mapcar #'first args)))
      ,@body))

#|
(defmacro -biggest-key-value (sequence &optional (key #'identity))
   "Use this for arrays.  Use < biggest-key-val > for lists."
  `(reduce #'max ,sequence :key ,key))

(defmacro -cap1 (string)
   `(string-upcase ,string :start 0 :end 1))

(defmacro -half (x)                    ; could be speed optimized, I'm sure.
  `(floor ,x 2))

(defmacro -kwote (form)
   `(list 'quote ,form))

(defmacro -the-last (list)
   `(car (last ,list)))
|#
;;; mm1 (991222) first version in heracd directory.  Prior was in oou.

@@file masterpick

(defun search-view-html (thread search)
  (let ((tally (search-tally search)))
    (setf (tally-wts tally) (default-pfchoice 'wts)
          (tally-geo tally) (default-pfchoice 'geo)
          (tally-srt tally) (default-pfchoice 'srt)
          (tally-fmt tally) (default-pfchoice 'fmt)
          (tally-max tally) (default-pfchoice 'max))
    (vt1 thread tally)))

;;; ------------------------------------------------------------------------
-----------|

(defparameter *sta-key* 'STA) ; used in VT lambda list
(defparameter *default-sta* 0) ; Lispy -- start counting from zero

(defcgifn vt (tk wts geo srt fmt max sta) ; sta = *sta-key*
  ;; vt = view thing
  ;; tk = tally key
  ;; wts = what to show
  ;; geo = geography filter
  ;; srt = sort-by criterion
  ;; fmt = format for display
  ;; max = max to show
  (let ((tally (tally-from-tallykey (read-from-string tk))))
    (setf (tally-wts tally) (get-pickfield-choice 'wts (read-from-string
wts))
          (tally-srt tally) (get-pickfield-choice 'srt (read-from-string
srt))
          (tally-fmt tally) (get-pickfield-choice 'fmt (read-from-string
fmt))
          (tally-max tally) (get-pickfield-choice 'max (read-from-string
max))
          (tally-geo tally) (or (get-pickfield-choice 'geo (read-from-string
geo))
                                (new-geo-pfchoice tally geo (read-from-string
geo)))
```

```
          (tally-sta tally) (if (zerop (length sta)) *default-sta* (read-
from-string sta))
          )
    (vt1 thread tally)))

(defhtmfn vt1 (thread tally) :props nil
  ;; Cannot make the assumption that a pfchoice will always be a member of
  ;; pickfield choices! (search-all-files on this comment to see why)
  ;;
  (with-new-page (thread :title (format nil "~A * ~A * ~A"
                                        (tally-filename tally)
                                        (pfchoice-name (tally-geo tally))
                                        (pfchoice-name (tally-wts tally))))
    (princt "<table cellpadding='5' width='100%'><tr><td bgcolor='99cc99'>"
thread)
    (breadcrumb          thread tally (tally-geo tally))
    (emit-pickfield-area thread tally)
    (princt "</td></tr></table>" thread)
    (emit-pagemeat-area thread tally)))

(defmethod new-geo-pfchoice (tally geostring (node-num integer))
  (declare (ignore geostring))
  (let* ((node      (node-from-node-num tally node-num))
         (name      (format nil "in ~A" (node-print-name node)))
         (nodeplace (node-place node)))
    (make-geonode-pfchoice :pickfieldkey 'GEO
                           :key    node-num
                           :name   name
                           :name2  name
                           :filter (if (unclassified-node-p node)
                                       #'(lambda (p) (place2-is-or-isin-place1
                                                      (place-superplace
nodeplace) p))
                                       #'(lambda (p) (place2-is-or-isin-place1
nodeplace p))))))

(defmethod new-geo-pfchoice (tally authorname garbage)
  (declare (ignore tally garbage))
  (make-geoname-pfchoice :pickfieldkey 'GEO
                         :key    (escape-author-name authorname)
                         :name   authorname
                         :name2  (format nil "of ~A" authorname)))

(defun geonode-pfchoice-node (pfc tally)
  (node-from-node-num tally (pfchoice-key pfc)))

;;; ---------------------------------------------------------------------------
-----------|

(defun emit-pagemeat-area (thread tally)
  (cond
   ((fmt-shows-debug-p tally)
    (if (geoname-pfchoice-p (tally-geo tally))
      (formatt thread "Nope.")
      (output-by-paper thread tally)))
   ;;
   ((geoname-pfchoice-p (tally-geo tally))
    (explode-author thread tally (pfchoice-name (tally-geo tally))))
   ;;
   ((wts-is-people-p tally)
    (print-authors thread tally))
```

```
    ;;
    ((wts-is-papers-p tally)
     (print-papers thread tally))
    ;;
    (t
     (let* ((wts (tally-wts tally))
            (geo (tally-geo tally))
            (dispnodes (findnodes  thread
                                   (pfchoice-filter wts)
                                   (pfchoice-filter geo)
                                   nil
                                   (node-subs
                                    (findnode-startnode geo tally)))))
       (if (< (length dispnodes) 2)
         (progn
           (setf (tally-wts tally) *paper-wtc-pfchoice*)
           (vtl thread tally))
         (emit-pagemeat-area1 thread tally
                              (sort dispnodes (pfchoice-filter (tally-srt
tally)))
                              wts))))))

(defmethod findnode-startnode (pfc tally)
  (declare (ignore pfc))
  (tally-root-node tally))

(defmethod findnode-startnode ((pfc geonode-pfchoice) tally)
  (geonode-pfchoice-node pfc tally))

(defparameter *min-display-score* 1)

(defun findnodes (thread wts-filter geo-filter toshow nodes)
  (if (endp nodes)
    (values toshow)
    (symbol-macrolet ((node1 (first nodes))
                      (place1 (node-place (first nodes))))
      (cond ((null (funcall wts-filter place1))
             (findnodes thread wts-filter geo-filter
                        (findnodes thread wts-filter geo-filter toshow (node-
subs node1))
                        (rest nodes)))
            ((or (null (funcall geo-filter place1))
                 (< (node-total-score node1) *min-display-score*))
             (findnodes thread wts-filter geo-filter toshow (rest nodes)))
            (t
             (findnodes thread wts-filter geo-filter (cons node1 toshow)
(rest nodes)))))))

(defparameter *number-sublink-nodes* 3)

(defun emit-pagemeat-area1 (thread tally nodes wts)
  (with-thread-output (stream thread)
    (let* ((worldwide-max-score (worldwide-max-score tally wts))
           (mag (calculate-score-bar-magnification nodes worldwide-max-
score))
           (min-display-score *min-display-score*)) ; could be parameterized
someday
      (flet ((1node (node)
               (when (>= (node-total-score node) min-display-score)
                 (format stream "~%<tr><td align='right'>")
                 (score-bar thread mag (node-total-score node))
```

```
                (princ "</td><td>" stream)
                (node-link thread tally node 1)
                (ecase (pfchoice-key (tally-fmt tally))
                  (sl (princ "</td></tr>" stream))
                  (sp (princ "</td><td>" stream)
                      (node-link-subs thread tally node 1 *number-sublink-
nodes*)
                      (princ "</td></tr>" stream))
                  (tr (princ "</td></tr><tr><td></td><td>" stream)
                      (node-link-subtree thread tally node 1 #'institution-
p)
                      (princ "</td></tr>" stream))))))
        (format stream "~%<table>")
        (map nil #'1node nodes) ; < nodes > may be list or vector
        (format stream "~%</table>")
        (worldwide-bar thread mag worldwide-max-score (pfchoice-name2
wts))))))

(defun worldwide-bar (thread mag worldwide-max-score name)
  (formatt thread "~%<table><tr><td valign='middle'>")
  (score-bar thread mag worldwide-max-score)
  (formatt thread "</td><td> &lt;-- Max score for ~A
worldwide.</td></tr></table>"
           name))

;;; ---------------------------------------------------------------------------
-----------|

(defun explode-author (thread tally authorname)
  (let* ((au (get-author-rec tally authorname))
         (papers (author-allpapers au))
         (leafplaces (author-leafplaces au)))
  (with-thread-output (stream thread)
    (princ "<blockquote>" stream)
    ;;
    ;; Name
    ;;
    (format stream "~%<b><big>~A</big></b>" authorname)
    ;;
    ;; Location(s)
    ;;
    (format stream "~%<p>Location(s):<ul>")
    (if leafplaces
      (dolist (p leafplaces)
        (princ "<li> " stream)
        (breadcrumb-up thread tally p t))
      (princ "<li> None could be determined." stream))
    (format stream "~%</ul>")
    ;;
    ;; (maybe) email address
    ;;
    (let ((emails (remove-if #'null
                             (mapcar #'paper-email (author-firstauthorpapers
au)))))
      (when emails
        (format stream "~%<p>Possible email address(es):<ul>~
                        ~{~%<li> <a href='mailto:~A'>~:*~A</a>~}~
                        ~%</ul>"
                (remove-duplicates emails :test #'string-equal))))
    ;;
    ;; Coauthors
```

```
    ;;
    (format stream "~%Co-authors, with number of co-authored
papers:<blockquote>")
    (let ((coauthors nil))
      (dolist (paper papers)
        (dolist (coauthor (paper-allauthors paper))
          (unless (eql coauthor au)
            (let ((rec (find coauthor coauthors :key #'first)))
              (if rec
                (incf (second rec))
                (push (list coauthor 1) coauthors))))))
      (if (null coauthors)
        (princ "None." stream)
        (progn
          (setq coauthors (sort coauthors #'> :key #'second))
          (person-link thread tally (caar coauthors))
          (format stream " (~D)" (second (first coauthors)))
          (dolist (co (rest coauthors))
            (format stream " &#183; ")
            (person-link thread tally (first co))
            (format stream " (~D)" (second co))))))
    (format stream "~%</blockquote>")
    ;;
    (format stream "~%<hr>Papers:")
    (print-papers thread tally (coerce papers 'vector))
    (princ "</blockquote>" stream))))

;;; ---------------------------------------------------------------------------
-----------|

(defun print-papers (thread
                     tally
                     &optional
                     (papers
                      (sort (papers-to-show tally (pfchoice-filter (tally-geo
tally)))
                            #'> :key #'paper-score)))
  (let ((len    (length papers))
        (maxx   (reduce #'max (tally-paperdata tally) :key #'paper-score))
        (start  (tally-sta tally))
        mag)
    (setf (tally-sta tally) *default-sta*) ; to help optimization in NODE-
LINK
    (setq mag (calculate-score-bar-magnification
               nil maxx :max-score-to-display (reduce #'max papers :key
#'paper-score)))
    (with-thread-output (stream thread)
      (format stream "~%<table>")
      (dotimes (i (things-per-page-bar thread stream tally len "Papers"
start))
        (let* ((paper (aref papers (+ i start)))
               (leafplaces (paper-leafplaces paper)))
          ;;
          ;; score bar and title line.
          ;;
          (format stream "~%<tr><td align='right' valign='top'>")
          (score-bar thread mag (paper-score paper))
          (format stream "</td><td>~A (~D)" (paper-title paper) (paper-year
paper))
          (html-image-tag thread "pm.gif" :border "0")
          (format stream "</td></tr>" (paper-title paper))
```

```
          ;;
          (format stream "~%<tr><td></td><td>")
          (dolist (name (paper-authornames paper))
            (person-link thread tally (get-author-rec tally name))
            (princ "  " stream))
          (format stream "</td></tr>")
          ;;
          (format stream "~%<tr><td></td><td>-A</td></tr>" (paper-address
paper))
          ;;
          ;; Breadcrumbing line
          ;;
          (format stream "~%<tr><td></td><td>")
          (dolist (p leafplaces)
            (princ "<small>[ " stream)
            (breadcrumb-up thread tally p t)
            (princ " ] </small>" stream))
          (format stream "</td></tr>")
          ;;
          ;; Separator bar
          ;;
          (format stream "~%<tr><td colspan='2'><hr></td></tr>")
          ))
      (format stream "</table>")
      (worldwide-bar thread mag maxx "a paper")
      (things-per-page-bar thread stream tally len "Papers" start))))

;;; ---------------------------------------------------------------------------
-----------|

(defun print-authors (thread tally)
  (let* ((authors  (find-authors tally (pfchoice-filter (tally-geo tally))))
         (len      (length authors))
         (maxx     0)
         (start    (tally-sta tally))
         mag)
    (setf (tally-sta tally) *default-sta*) ; to help optimization in NODE-
LINK
    (maphash #'(lambda (key author-rec)
                 (declare (ignore key))
                 (setq maxx (max maxx (author-score author-rec))))
             (tally-authordata tally))
    (setq mag (calculate-score-bar-magnification
               nil maxx :max-score-to-display (reduce #'max authors :key
#'author-score)))
    (with-thread-output (stream thread)
      (format stream "~%<table>")
      (dotimes (i (things-per-page-bar thread stream tally len "People"
start))
        (let ((au (aref authors (+ i start))))
          ;;
          ;; score bar and author link
          ;;
          (format stream "~%<tr><td align='right'>")
          (score-bar thread mag (author-score au))
          (format stream "</td><td>")
          (person-link thread tally au)
          (format stream "</td>")
          ;;
          ;; location breadcrumbing
          ;;
```

```
          (multiple-value-bind (leafplaces certaintylevel)
                               (author-leafplaces au)
            (case certaintylevel
              (1 (breadcrumb-authorplaces-in-cell thread stream tally
leafplaces nil))
              (2 (breadcrumb-authorplaces-in-cell thread stream tally
leafplaces t))
              (3 (format stream
                         "<td><small>??</small></td><td><small>~{[ ~:(~A~)
]~}</small></td>"
                         leafplaces))
              (4 (princ "<td><small>?</small></td><td><small>?</small></td>"
stream))))
          ;;
          ;; end table row
          ;;
          (princ "</tr>" stream)))
      (format stream "</table>")
      (worldwide-bar thread mag maxx "a person"))))

(defun author-leafplaces (au)
  ;; Returns two values:
  ;; (a) a list of leafplaces, which may be NIL.
  ;; (b) the "category" of the leafplace answer.
  ;;  1=got places from papers on which author was first author,
  ;;  2=got places from papers on which author was not first author,
  ;;  3=got places from the CY fields of all the authors papers,
  ;;  4=no places.
  (let ((leafplaces (find-leafplaces-for-papers (author-firstauthorpapers
au))))
    (if leafplaces
      (values leafplaces 1)
      (if (setq leafplaces (find-leafplaces-for-papers (author-allpapers
au)))
        (values leafplaces 2)
        (if (setq leafplaces (mapcar #'paper-country (author-allpapers au)))
          (values (remove-duplicates leafplaces :test #'string-equal) 3)
          (values nil 4))))))

(defun breadcrumb-authorplaces-in-cell (thread stream tally leafplaces
questionp)
  (princ (if questionp
           "<td><small>?</small></td>"
           "<td></td>")
         stream)
  (princ "<td><small>" stream)
  (dolist (place leafplaces)
    (princ "[ " stream)
    (breadcrumb-up thread tally place t)
    (princ " ] " stream))
  (princ "</small></td>" stream))

(defun find-authors (tally geo-filter)
  (let ((authors (make-array 1000 :adjustable t :fill-pointer 0)))
    (dovec (paper i (papers-to-show tally geo-filter))
      (dolist (author-rec (paper-allauthors paper))
        (when (not (find author-rec authors))
          (vector-push-extend author-rec authors))))
    (sort authors #'> :key #'author-score)))

(defun author-details (stream au)
```

```
  (format stream
          "<p><tt>~4,'0D ~D ~D ~D </tt><a
href='javascript:pm(\"~A~{,~A~}\")'>~A</a>"
          (author-score au)
          (length (author-allpapers au))
          (length (author-firstauthorpapers au))
          (length (author-lastauthorpapers au))
          (paper-pmid (first (author-allpapers au)))
          (mapcar #'paper-pmid (rest (author-allpapers au)))
          (author-name au)))

(defun paper-details (stream au address-papers)
  (dolist (paper (sort (copy-seq (author-allpapers au)) #'> :key #'paper-
year))
    (format stream "~%<br><tt>    ~A </tt>~A (~D)"
            (if (member paper address-papers :test #'eql) "*" " ")
            ; (paper-score paper)
            (paper-title paper)
            (paper-year paper))))

;;; ------------------------------------------------------------------------
-----------|
;;; This is "debug" output

(defun output-by-paper (thread tally)
  (output-by-paper2 thread tally 0))

(defcgifn output-by-paper1 (tallykey start)
  (output-by-paper1h thread
                     (tally-from-tallykey (read-from-string tallykey))
                     (read-from-string start)))
(defhtmfn output-by-paper1h (thread tally start) :props nil
  (with-new-page (thread :title (tally-filename tally))
    (output-by-paper2 thread tally start)))

(defun output-by-paper2 (thread tally start)
  (let* ((papers          (papers-to-show tally (pfchoice-filter (tally-geo
tally))))
         (len             (length papers)))
    (with-thread-output (stream thread)
      (dotimes (i (paper-counter-bar thread stream tally len start))
        (let ((paper (aref papers (+ i start))))
          (format stream "~%<hr><tt>~7,2F </tt>~A [~A]~%<ul>"
                  (paper-year paper)
                  (or (paper-address paper) "-- no address given --")
                  (paper-country paper))
          (format stream "~%<br>Email address: ~A" (paper-email paper))
          (dolist (rule (paper-rules paper))
            (format stream "~%<li> <small>")
            ;THIS IS GIVING A PROBLEM
            (node-link thread tally (find-place-node (rule-place rule) tally)
0)
            (format stream " ** ~S</small>" rule))
          (princ "</ul>" stream)))
      (paper-counter-bar thread stream tally len start))))

(defun papers-to-show (tally geo-filter)
  (let ((papers (make-array 1000 :adjustable t :fill-pointer 0)))
    (dovec (paper i (tally-paperdata tally))
      (dolist (leafplace (paper-leafplaces paper))
```

```
        ;(dbp (place-key leafplace) (if (funcall geo-filter leafplace) t
nil))
        (when (funcall geo-filter leafplace)
          (vector-push-extend paper papers)
          (return))))
    (values papers)))

(defun paper-counter-bar (thread streem tally npapers start)
  (let* ((papers-per-page (pfchoice-filter (tally-max tally)))
         (n (min papers-per-page (- npapers start))))
    (format streem "~%<hr>Papers ~D--D." (1+ start) (+ start n))
    (dotimes (i (ceiling npapers papers-per-page))
      (let ((j (* i papers-per-page)))
        (princ "   " streem)
        (unless (= start j)
          (cgi-anchor thread 'OUTPUT-BY-PAPER1
                      'tallykey (tally-key tally)
                      'start j))
        (format streem "~D--D" (1+ j) (min npapers (+ j papers-per-page)))
        (unless (= start j)
          (princ "</a>" streem))))
    (values n)))

(defun things-per-page-bar (thread streem tally nthings thingsname start)
  (let* ((things-per-page (pfchoice-filter (tally-max tally)))
         (n               (min things-per-page (- nthings start))))
    (unless (and (zerop start)
                 (= n nthings))
      (format streem "~%<hr><b>~A ~D--D</b>" thingsname (1+ start) (+ start
n))
      (dotimes (i (ceiling nthings things-per-page))
        (let* ((j     (* i things-per-page))
               (group (format nil "~D--D"
                              (1+ j) (min nthings (+ j things-per-page)))))
          (princ " &#183; " streem)
          (if (= start j)
            (princ group streem)
            (start-link  thread tally group j))))
      (princ "<hr>" streem))
    (values n)))

;;; ---------------------------------------------------------------------
-----------|

(defun worldwide-max-score (tally wts)
  (let ((nodes (tally-nodes tally))
        (thing-filter (pfchoice-filter wts)))
    (if (plusp (length nodes))
      (reduce #'max nodes :key (lambda (node)
                                 (if (and (funcall thing-filter (node-place
node))
                                          (not (unclassified-node-p node)))
                                   (node-total-score node)
                                   0)))
      1)))

(defun calculate-score-bar-magnification
       (display-nodes worldwide-max-score
                      &key (max-score-to-display (reduce #'max display-nodes
                                                         :key #'node-total-
score)))
```

```
  ;; Reduce the magnification if the worldwide bar would be too huge.
  (let* ((max-bar-width 70)
         (mag (/ max-bar-width max-score-to-display)))
    (when (> (* mag worldwide-max-score) 450)
      (setq mag (/ 450 worldwide-max-score)))
    (values mag)))

(defun score-bar (thread mag score)
  (let ((blackwidth (max 1 (round (* mag score)))))
    (html-image-tag thread "1x1black.gif" :width blackwidth :height 8)))

;;; ---------------------------------------------------------------------------
-----------|

(defun node-link (thread tally node nlevels)
  (DECLARE (IGNORE NLEVELS))
  (declare (special *window-root-node-key*))
  (with-slots (wts geo srt fmt max sta) tally
    (cgi-anchor-with-text thread (node-print-name node)
                          'VT
                          'TK (tally-key tally)
                          (pfchoice-pickfieldkey wts) (wts-rightlevel node)
                          (pfchoice-pickfieldkey geo) (node-number node)
                          (pfchoice-pickfieldkey srt) (pfchoice-key srt)
                          (pfchoice-pickfieldkey fmt) (pfchoice-key fmt)
                          (pfchoice-pickfieldkey max) (pfchoice-key max))))

(defun start-link (thread tally text startnum)
  (declare (special *window-root-node-key*))
  (with-slots (wts geo srt fmt max sta end) tally
      (cgi-anchor-with-text thread text
                            'VT
                            'TK (tally-key tally)
                            (pfchoice-pickfieldkey wts) (pfchoice-key wts)
                            (pfchoice-pickfieldkey geo) (pfchoice-key geo)
                            (pfchoice-pickfieldkey srt) (pfchoice-key srt)
                            (pfchoice-pickfieldkey fmt) (pfchoice-key fmt)
                            (pfchoice-pickfieldkey max) (pfchoice-key max)
                            *sta-key*                   startnum)))

(defun person-link (thread tally author-rec)
  (declare (special *window-root-node-key*))
  (let ((name (author-name author-rec)))
    (with-slots (wts geo srt fmt max sta end) tally
      (cgi-anchor thread
                  'VT
                  'TK (tally-key tally)
                  (pfchoice-pickfieldkey wts) (pfchoice-key *paper-wtc-
pfchoice*)
                  (pfchoice-pickfieldkey geo) (escape-author-name name)
                  (pfchoice-pickfieldkey srt) (pfchoice-key srt)
                  (pfchoice-pickfieldkey fmt) (pfchoice-key fmt)
                  (pfchoice-pickfieldkey max) (pfchoice-key max)
                  *sta-key*                   0)
      (princt name thread)
      (html-image-tag thread "pm.gif" :border "0")
      (princt "</a>" thread))))

(defun node-link-subs (thread tally node nlevels nprint)
  (let ((subs (sort (remove-if #'zerop (node-subs node) :key #'node-total-
score)
```

```
                    (pfchoice-filter (tally-srt tally)))))
    (when subs
      (formatt thread "[<small> ")
      (node-link thread tally (first subs) (1+ nlevels))
      (dolist (sub (rest subs))
        (when (zerop (decf nprint))
          (return))
        (princt " * " thread)
        (node-link thread tally sub (1+ nlevels)))
      (when (nthcdr nprint subs)
        (princt " ..." thread))
      (princt " </small>]" thread))))

(defun node-link-subtree (thread tally node nlevels leaf-test-fn)
  (let ((subs (sort (remove-if #'zerop (node-subs node) :key #'node-total-
score)
                    (pfchoice-filter (tally-srt tally)))))
    (dolist (sub subs)
      (unless (and (= nlevels 1)
                   (eq sub (first subs)))
        (formatt thread "~%<br>"))
      (dotimes (i nlevels)
        (formatt thread "     "))
      (node-link thread tally sub nlevels)
      (unless (funcall leaf-test-fn (node-place sub))
        (node-link-subtree thread tally sub (1+ nlevels) leaf-test-fn)))))

;;; ------------------------------------------------------------------------
-----------|

(defun emit-pickfield-area (thread tally)
  (cgi-form-start thread 'VT)
  (cgi-form-hidden thread
                   'TK (tally-key tally))
  (apply #'emit-pickfield-menu 'wts thread (calc-pfcs-wts (tally-wts tally)))
  (apply #'emit-pickfield-menu 'geo thread (calc-pfcs-geo (tally-geo tally)))
  (apply #'emit-pickfield-menu 'srt thread (calc-pfcs     (tally-srt tally)))
  (formatt thread "<input type='submit' value='Go'>~%<br>~%")
  (apply #'emit-pickfield-flat 'fmt thread (calc-pfcs     (tally-fmt tally)))
  (apply #'emit-pickfield-menu 'max thread (calc-pfcs     (tally-max tally)))
  (princt "</form>" thread))

(defun world-p (place)
  (placekey= (place-key place)
             *world-key*))

#| This fn was neat. If NYC was the window's underlying place, then
   this fn would prevent you from asking for countries in NYC.
   But it would also prevent you from asking for countries in the
   world, so I replaces it with vanilla < calc-pfcs >.
(defun calc-pfcs-wts (pfchoice place)
  (let ((nix (etypecase place
               (PLANET      nil)
               (COUNTRY     '(oc))
               (STAYT       '(oc ts))
               (CITY        '(oc ts ic))
               (INSTITUTION '(oc ts ic ni))
               (PEOPLE      '(oc ts ic ni ep)))))
    (list (remove-if #'(lambda (pfc-key)
                         (member pfc-key nix))
                     (built-in-pfchoices pfchoice)
```

```
                    :key #'pfchoice-key)
          pfchoice)))
|#

(defun calc-pfcs-wts (pfchoice) ; see above
  (calc-pfcs pfchoice))

(defmethod calc-pfcs-geo (pfchoice)
  (calc-pfcs pfchoice))

(defmethod calc-pfcs-geo ((pfc geonode-pfchoice))
  (list (cons pfc (built-in-pfchoices pfc))
        pfc))

(defmethod calc-pfcs-geo ((pfc geoname-pfchoice))
  (list (cons pfc (built-in-pfchoices pfc))
        pfc))

(defun calc-pfcs (pfchoice)
  ;; Use this fn for pfchoices that never vary, e.g. sort, fmt, max
  (list (built-in-pfchoices pfchoice)
        pfchoice))

(defun built-in-pfchoices (pfchoice)
  (pickfield-choices (pickfield-from-key (pfchoice-pickfieldkey pfchoice))))

(defun emit-pickfield-menu (pickfield-key thread pfcs selected-pfc)
  (with-thread-output (stream thread)
    (format stream "<select name='~S'>" pickfield-key)
    (dolist (pfc pfcs)
      (format stream "<option value=~S~A>~A"
              (pfchoice-key pfc)
              (if (eql pfc selected-pfc) " SELECTED" "")
              (pfchoice-name2 pfc)))
    (format stream "~%</select>")))

(defun emit-pickfield-flat (pickfield-key thread pfcs selected-pfc)
  (with-thread-output (stream thread)
    (format stream "~A:" (pickfield-name (pickfield-from-key pickfield-key)))
    (dolist (pfc pfcs)
      (format stream "  <input type='radio' name='~S' value='~S'~A> ~A
 "
              pickfield-key
              (pfchoice-key pfc)
              (if (eql pfc selected-pfc) " CHECKED" "")
              (pfchoice-name2 pfc)))))

;;; ---------------------------------------------------------------------
-----------|

(defparameter *breadcrumb-arrow* " &gt; ")


(defun breadcrumb (thread tally geo)
  (declare (special *world-geo-pfchoice*))
  (with-thread-output (stream thread)
    ;;
    ;; Emit the name of the search as first breadcrumb
    ;;
    (let ((searchname (search-name-from-tally-filename (tally-filename
tally))))
```

```
      (cgi-anchor-with-text thread searchname 'SEARCHHOME
                            'id searchname))
    (princ *breadcrumb-arrow* stream)
    ;;
    ;; Emit the super-places, as links
    ;;
    (breadcrumb-place thread tally (breadcrumb1 geo tally))
    ;;
    ;; Emit current place, as plaintext (not a link)
    ;;
    (format stream (pfchoice-name geo))
    ;;
    ;; Maybe output a www address
    ;;
    (let (x)
      (when (and (geonode-pfchoice-p geo)
                 (setq x (node-place (geonode-pfchoice-node geo tally)))
                 (institution-p x)
                 (setq x (institution-www x)))
        (formatt thread "     <a href='~A' target=_new>~:*~A</a>"
x)))))

(defmethod breadcrumb1 ((pfc geonode-pfchoice) tally)
  (place-superplace (node-place (geonode-pfchoice-node pfc tally))))

(defmethod breadcrumb1 ((pfc geoname-pfchoice) tally)
  (multiple-value-bind (places level)
                       (author-leafplaces (get-author-rec tally (pfchoice-
name pfc)))
    (if (<= level 2)
      (first places)
      nil)))

(defmethod breadcrumb1 (pfc tally) ; members of std geo menu
  (declare (ignore pfc tally))
  (top-place))

(defun breadcrumb-place (thread tally place)
  (when place
    (breadcrumb-place thread tally (place-superplace place))
    (node-link thread tally (find-place-node place tally) 1)
    (formatt thread *breadcrumb-arrow*)))

(defun breadcrumb-up (thread tally place firstp)
  (unless (top-place-p place)
    (when (not firstp)
      (princt " &#183; " thread))
    (node-link thread tally (find-place-node place tally) 1)
    (breadcrumb-up thread tally (place-superplace place) nil)))


@@file noder

(defstruct node
  number
  place
  (score  0)
  (score? 0)     ; when there is >1 leafplace for a paper
  (papers  nil)
  (papers? nil) ; when there is >1 leafplace for a paper
  (subs nil)
```

```
  )

(defstruct (unclassified-node (:include node))
  )

(defstruct (person-node (:include node))
  authorscore)

;;; ---------------------------------------------------------------------------
-----------|

(defun make-nodes (tally)
  (let* ((nodes (make-array 100 :adjustable t :fill-pointer 0))
         (top-place (place-from-place-key *world-key*))
         (top-node (add-a-node nodes (make-node :place top-place))))
    (make-nodesx nodes top-node (place-subkeys top-place))
    (setf (tally-nodes tally) nodes)
    (setf (tally-topnode tally) top-node)
    (values)))

(defun add-a-node (nodes node)
  (setf (node-number node) (fill-pointer nodes))
  (vector-push-extend node nodes)
  (values node))

(defun find-place-node (place tally)
  (find place (tally-nodes tally)
        :test #'(lambda (place node)
                  (and (eql place (node-place node))
                       (not (unclassified-node-p node))))))

(defun node-from-place-key (placekey tally)
  (find-place-node (place-from-place-key placekey) tally))

(defun node-from-node-num (tally num)
  (aref (tally-nodes tally) num))

(defun tally-root-node (tally)
  (node-from-place-key *world-key* tally))

(defun make-nodesx (nodes node subplace-keys)
  (when subplace-keys
    (let* ((place1 (place-from-place-key (first subplace-keys)))
           (node1  (add-a-node nodes (make-node :place place1))))
      (push node1 (node-subs node))
      (make-nodesx nodes node1 (place-subkeys place1))
      (make-nodesx nodes node (rest subplace-keys)))))

(defun node-total-score (node)
  (+ (node-score  node)
     (node-score? node)))

(defun score-nodes (tally)
  ;; First, assign each paper to one or more nodes.
  ;; If a paper has no place assigned (ie, no leafplaces) assign it to the
world node.
  (let ((nodes (tally-nodes tally))
        leafplaces
        papernode)
    (dovec (paper i (tally-paperdata tally))
      (setq papernode (if (setq leafplaces (paper-leafplaces paper))
```

```
                        (find (first leafplaces) nodes :key #'node-place)
                        (tally-topnode tally)))
      (push paper (node-papers papernode))
      (incf (node-score papernode) (paper-score paper))
      (dolist (place (rest leafplaces))
        (setq papernode (find place nodes :key #'node-place))
        (push paper (node-papers? papernode))
        (incf (node-score? papernode) (paper-score paper)))
      )
    ;; Now compute the scores for the node tree
    (score-nodesx nodes (tally-topnode tally)))
  (values))

(defun score-nodesx (nodes node)
  (dolist (subnode (node-subs node))
    (score-nodesx nodes subnode))
  (when (and (node-subs node)
             (or (node-papers  node) ; Why is this OR clause here?
                 (node-papers? node))
             (find-if #'plusp (node-subs node) :key #'node-total-score))
    (push (add-a-node nodes
                      (make-unclassified-node
                       :place   (find-unclassified-place
                                 (node-place node)
                                 (node-place (first (node-subs node))))
                       :papers  (node-papers  node)
                       :papers? (node-papers? node)
                       :score   (node-score  node)
                       :score?  (node-score? node)
                       :number (fill-pointer nodes)))
          (node-subs node))
    (setf (node-papers  node) nil
          (node-papers? node) nil
          (node-score  node) 0
          (node-score? node) 0))
  ;; Now can do simple summations
  (incf (node-score  node) (reduce #'+ (node-subs node) :key #'node-score))
  (incf (node-score? node) (reduce #'+ (node-subs node) :key #'node-score?))
  (values))

(defun alpha-countries (p1 p2)
   (cond ((eql (place-key p1) 'US)
          t)
         ((eql (place-key p2) 'US)
          nil)
         ((string-lessp (place-name p1) (place-name p2))
          t)
         (t
           nil)))

;;; (defun alpha-place (p1 p2)
;;;    (let ((diff (- (position (type-of p1) '(COUNTRY STAYT CITY
INSTITUTION))
;;;                   (position (type-of p2) '(COUNTRY STAYT CITY
INSTITUTION)))))
;;;       (cond ((plusp diff)
;;;              p1)
;;;             ((minusp diff)
;;;              p2)
;;;             ((equal (place-key p1) '(co . us))
;;;              p1)
```

```
;;;             ((equal (place-key p2) '(co . us))
;;;              p2)
;;;             ((string-lessp (place-name p1) (place-name p2))
;;;              p1)
;;;             (t
;;;               p2))))

@@file parse-paper

(defstruct field
  key
  value
  )

#|
(defun country-name-to-abbrev (name)
  (or (second (find name '(("UNITED STATES" US)
                           ("ENGLAND" UK)
                           ("INDIA" IN)
                           ("NETHERLANDS" NL)
                           ("ITALY" IT)
                           ("DENMARK" DK)
                           ("IRELAND" IE)
                           ("GERMANY" DE)
                           ("FRANCE" FR)
                           )
                     :key #'first
                     :test #'string=))
      (progn
        (dbp 'unfound-country name)
        name)))
|#

(let ((fields-of-interest '(AU CY AD PMID- PT DP TI)))

  (defun parse-paper (f outfile)
    ;; Last author is processed first.
    (let (pmid titlelines authors pubtypes addresslines year country)
      (dolist (field (extract-fields f))
        (ecase (field-key field)
          (AU    (push (field-value field) authors))
          (CY    (setf country (field-value field)))
          (AD    (push (field-value field) addresslines))
          (PMID- (setf pmid (field-value field)))
          (TI    (push (field-value field) titlelines))
          (PT    (push (field-value field) pubtypes))
          (DP    (setf year (derive-paper-date (field-value field))))
          ))
      ;;
      ;; Write the paper to the LXL file
      ;;
      (format outfile "~&(make-paper")
      (format outfile "~@{~%  ~S ~S~}"
              :pmid        pmid
              :title       (format nil "~A~{ ~A~}" (first titlelines) (rest
titlelines))
              :authornames (cons 'LIST (reverse authors))
              :lastauthor  (first authors) ; is actually last author
              :authorcount (length authors)
              :pt          (cons 'LIST pubtypes)
```

```
                :address      (format nil "~A~{ ~A~}" (first addresslines) (rest
addresslines))
                :year         (or year 0)
                :country      country
                )
        (format outfile ")")
        (values)))

    (defun extract-fields (f)
       (let ((fieldrecs nil))
          (catch 'paper-has-ended
            (loop
              (if (paper-has-ended-p *line-buffer*)
                 (throw 'paper-has-ended t)
                 (let ((fieldkey (read-from-string *line-buffer*)))
                    (if (member fieldkey fields-of-interest)
                       (push (make-field :key fieldkey :value (get-field-value
f))
                             fieldrecs)
                       (read-until-next-field-start f))))))
          ;(dbp 'field-recs= fieldrecs)
          (values fieldrecs))))

(defun derive-paper-date (yr+mo)
  ;; < yr+mo > is the value of the Medline DP field. Format: "1993 Jun"
  ;; This is changed to a floating point number, eg: 1993.06
  ;; If we cannot discover a date, it's set to 0000.
  (multiple-value-bind (yr restart)
                       (read-from-string yr+mo)
    (unless (numberp yr)
      (unless (numberp (setq yr (read-from-string yr+mo nil nil
                                                  :end (min 4 (length
yr+mo)))))
        (setq yr 0000)))
    (let ((mo (position (read-from-string yr+mo nil nil
                                          :start restart
                                          :end (min (+ 3 restart) (length
yr+mo)))
                        '(xx jan feb mar apr may jun jul aug sep oct nov
dec))))
      (when mo
        (incf yr (float (/ mo 100))))
      (values yr))))

(defun get-field-value (f)
   (let ((vals (list (subseq *line-buffer* 6))))
      (catch 'next-field
        (loop
          (read-a-line f)
          (when (plusp (length *line-buffer*))
             (if (char= (aref *line-buffer* 0) #\Space)
                (push (subseq *line-buffer* 5) vals) ; 5 leaves leading space
                (throw 'next-field t)))))
      ; (dbp (format nil "~{~A~}" (reverse vals)))
      (values (format nil "~{~A~}" (reverse vals)))))

(defun read-until-next-field-start (f)
   (catch 'next-field
     (loop
       (read-a-line f)
       (and (plusp (length *line-buffer*))
```

```
            (not (char= (aref *line-buffer* 0) #\Space))
            (throw 'next-field t)))))

(defun paper-has-ended-p (line)
   (search "</pre>" line :test #'char-equal))

(defun find-paper-start (f)
   ;; When it returns, *line-buffer* is the first AU line.
   ;; Ends up losing the UI line that precedes the first AU,
   ;; but we don't care.
   (let ((prevline ""))
      (catch 'found-paper-start++
        (loop
          (read-a-line f) ; might throw 'end-of-file
          (if (and (> (or (mismatch *line-buffer* "AU  - " :test #'char=)
                          -1)
                      5)
                   (search "<pre>" prevline :test #'char-equal))
              (throw 'found-paper-start++ 't)
              (setq prevline *line-buffer*)))
        (values))))

(let ((eof (gensym)))
   (defun read-a-line (f)
      (setq *line-buffer* (read-line f nil eof))
      (if (eql eof *line-buffer*)
         (throw 'end-of-file t)
         *line-buffer*)))


@@file pickfield

(defstruct pickfield
  name
  key
  choices
  )

(defstruct pfchoice
  pickfieldkey
  selectedp
  key
  name
  name2
  filter
  )

(defstruct (geonode-pfchoice (:include pfchoice)))
(defstruct (geoname-pfchoice (:include pfchoice)))

(defmacro make-a-pickfield (name key choice-specs)
  `(make-pickfield :name    ,name
                   :key     ,key
                   :choices ,choice-specs))

(defmacro make-a-pfchoice (pickfield-key selectedp key name flatname filter)
  `(make-pfchoice :pickfieldkey (quote ,pickfield-key)
                  :selectedp    ,selectedp
                  :key          (quote ,key)
                  :name         ,name
                  :name2        (or ,flatname ,name)
```

```
                    :filter       ,filter)) ; in case we need place for extra
data

;;; ---------------------------------------------------------------------------
-----------|

#|
(defun usa/canada-p (p)
  (or (place-placekey= p *place-usa*)
      (place-placekey= p *place-canada*)))
|#

(defvar *world-geo-pfchoice*)
(defvar *paper-wtc-pfchoice*)

(defparameter *pickfields*

  ;; Cannot make the assumption that a pfchoice will always be a member of
  ;; pickfield choices! (search-all-files on this comment to see why)

  (list

   (make-a-pickfield ; Choices must be in ascending order for < wts-1level-
down >
    "What to show"
    'wts
    ;; Must be synchronized with < defmethod place-pfchoice-key >
    (list (setq
           *paper-wtc-pfchoice*
           (make-a-pfchoice wts nil ap "Papers"           nil 'PAPER-P
))
          (make-a-pfchoice  wts nil ep "People"           nil #'FALSE
) ; stub
          (make-a-pfchoice  wts t   ni "Institutions"     nil #'institution-p
)
          (make-a-pfchoice  wts nil ic "Cities"           nil #'city-p
)
          (make-a-pfchoice  wts nil ts "States/Provinces" nil #'stayt-p
)
          (make-a-pfchoice  wts nil oc "Countries"        nil #'country-p
)))

   (make-a-pickfield
    "Geography"
    'geo
    (list (setq
           *world-geo-pfchoice*
           (make-a-pfchoice geo nil w "The world"     "in the world"
                            #'true))
          (make-a-pfchoice  geo nil u "USA/Canada"    "in USA/Canada"
                            #'(lambda (p)
                                (or (place2-is-or-isin-place1 *place-usa* p)
                                    (place2-is-or-isin-place1 *place-canada*
p))))
          (make-a-pfchoice  geo nil b "Britain/Ireland" "in Britain/Ireland"
                            #'(lambda (p)
                                (or (place2-is-or-isin-place1 *place-uk* p)
                                    (place2-is-or-isin-place1 *place-ireland*
p))))
          (make-a-pfchoice  geo t   e "US/Canada/Britain/Ireland" "in
US/UK/Can./Ire."
```

```
                              #'(lambda (p)
                                  (or (place2-is-or-isin-place1 *place-usa* p)
                                      (place2-is-or-isin-place1 *place-canada*
p)
                                      (place2-is-or-isin-place1 *place-uk* p)
                                      (place2-is-or-isin-place1 *place-ireland*
p))))
          (make-a-pfchoice  geo nil o "Outside the USA" "outside USA"
                              #'(lambda (p)
                                  (not (place2-is-or-isin-place1 *place-usa*
p))))))

   (make-a-pickfield
    "Sort by"
    'srt
    (list (make-a-pfchoice srt t   s "Score"                 "sorted by
score"
                              #'(lambda (node1 node2)
                                  (cond ((eql (type-of node1) (type-of node2))
                                         (> (node-total-score node1)
                                            (node-total-score node2)))
                                        ((unclassified-node-p node2)
                                         t)
                                        (t nil))))
          (make-a-pfchoice srt nil n "Name"                  "sorted by name"
                              #'(lambda (node1 node2)
                                  (cond ((eql (type-of node1) (type-of node2))
                                         (string-lessp (place-name (node-place
node1))
                                                       (place-name (node-place
node2))))
                                        ((unclassified-node-p node2)
                                         t)
                                        (t nil))))
          (make-a-pfchoice srt nil d "ZIPcode" "sorted by ZIPcode"
                              #'(lambda (node1 node2)
                                  (> (length (node-subs node1))
                                     (length (node-subs node2)))))
          (make-a-pfchoice srt nil d "Distance from ZIPcode" "sorted by
distance"
                              #'(lambda (node1 node2)
                                  (> (length (node-subs node1))
                                     (length (node-subs node2))))))))

   (make-a-pickfield
    "Max per page"
    'max
    (list (make-a-pfchoice max nil   10   "10" "show 10 max"    10)
          (make-a-pfchoice max t     30   "30" "show 30 max"    30)
          (make-a-pfchoice max nil  100  "100" "show 100 max"  100)
          (make-a-pfchoice max nil  all  "All" "show all"      999999))) ;
kludge


   (make-a-pickfield
    "Output format"
    'fmt
    (list (make-a-pfchoice fmt nil sl "Sublist"  "List"     'spn-sublist)
          (make-a-pfchoice fmt t   sp "Sublist+" "List+"    'spn-sublist+)
          (make-a-pfchoice fmt nil tr "Tree"     "Tree"     'spn-subtree)
          (make-a-pfchoice fmt nil db "Debug"    "*debug*" 'spn-papers+)
```

```
        ;(make-a-pfchoice fmt nil pa "Papers"   "Papers"  'spn-papers)
         ))
   ))

(defun pickfield-from-key (pickfield-key)
  (find pickfield-key *pickfields* :key #'pickfield-key))

(defun get-pickfield-choice (pickfield-key pfchoice-key)
  ;; NIL is an occasional, and useful, return value.
  ;; (e.g. when use menus to say I want to list all institutions in
California)
  (find pfchoice-key
        (pickfield-choices
         (find pickfield-key *pickfields* :key #'pickfield-key))
        :key #'pfchoice-key))

(defparameter *window-root-node-key* 'rw)
(assert (null (get-pickfield-choice 'GEO *window-root-node-key*)) nil
        "Cannot give *window-root-node-key* a keyvalue used by another GEO
pfchoice.")

(defun default-pfchoice (pickfield-key)
  (find-if #'pfchoice-selectedp (pickfield-choices (pickfield-from-key
pickfield-key))))

#|
(defun wts-downlevel (wts-pfchoice nlevels)
  (let* ((pfchoices (pickfield-choices
                     (pickfield-from-key
                      (pfchoice-pickfieldkey wts-pfchoice))))
         (pos (- (position wts-pfchoice pfchoices) nlevels)))
    (nth pos pfchoices)))
|#

(defun wts-rightlevel (node)
  (if (node-subs node)
    (place-pfchoice-key (node-place (first (node-subs node))))
    (etypecase (node-place node)
      (PLANET      'oc)
      (COUNTRY     'ts)
      (STAYT       'ic)
      (CITY        'ni)
      (INSTITUTION 'ep)
      (PEOPLE      'ap))))

(defun fmt-shows-papers-p (tally)
  (eql (pfchoice-key (tally-fmt tally))
       'PA))

(defun fmt-shows-debug-p (tally)
  (eql (pfchoice-key (tally-fmt tally))
       'DB))

(defun wts-is-people-p (tally)
  (eql (pfchoice-key (tally-wts tally))
       'EP))

(defun wts-is-papers-p (tally)
  (eql (pfchoice-key (tally-wts tally))
       'AP))
```

```
@@file places-find

(defun locate (address)
   (let ((places nil))
      (multiple-value-bind (tokens email-domains email-address)
          (tokenize-address (string-upcase address))
         (setq places (try-domain-rules places email-domains))
         (dotimes (i (fill-pointer tokens))
            (setq places (try-nontld-rules places tokens i (string-right-trim
",." (aref tokens i)))))
         (dotimes (i (fill-pointer tokens))
            (setq places (try-help-rules places tokens i (string-right-trim
",." (aref tokens i)))))
         (values places email-address))))

(defun try-domain-rules (places test-domains)
  (when test-domains
    (dolist (rule (gethash (first test-domains) *geo-tokens* nil))
      (when (domain-rule-p rule)
        (let* ((rule-domains (rule-parmdata rule))
               (diff (mismatch test-domains rule-domains :test #'string=)))
          (setq places
                (maybe-add-place places rule (or (null diff)
                                                 (>= diff (length rule-
domains))))))))))
  (values places))

(defun try-nontld-rules (places tokens i token)
  (dolist (rule (gethash token *geo-tokens* nil))
    (unless (member (rule-id rule) places :key #'rule-id :test #'=)
      (setq places
            (maybe-add-place places rule
                             (case (rule-method rule) ; return rule if rule
successful
                               (:one   t)
                               (:adj   (the-adj-test i tokens (rule-parmdata
rule)))
                               (:all   (the-all-test i tokens (rule-parmdata
rule)))
                               (t      nil))))))
  (values places))

(defun the-adj-test (address-i address-tokens rule-tokens)
   ;; returns T or NIL
   (when (<=
           (+ address-i (length rule-tokens))
           (length address-tokens))
      (let ((ii address-i)
            (matchp t))
         (dolist (rule-token rule-tokens)
            (if (string-equal rule-token (aref address-tokens ii))
               (incf ii)
               (return (setq matchp nil))))
         (values matchp))))

(defun the-all-test (address-i address-tokens rule-tokens)
   ;; returns T or NIL
   (let ((ii address-i)
         (okp t))
      (dolist (rule-token rule-tokens)
```

```
          (setq ii (position rule-token address-tokens :start ii :test
#'string-equal))
          (when (null ii)
             (setq okp nil)
             (return)))
       (values okp)))

(defun the-help-test (successful-rules where-type where-key)
  ;; Note that CLtL2 page 391 guarantees that the calls have the form:
  ;; (funcall #'place1-is-or-isin-place2 item (keyfn sequence-item))
  ;(dbp (place-key (place-from-2step where-type where-key))
  ;     (mapcar #'place-key (mapcar #'rule-place successful-rules)))
  ;(dolist (r successful-rules)
  ;  (princ " ")
  ;  (princ (if (place2-is-or-isin-place1
  ;              (place-from-2step where-type where-key)
  ;              (rule-place r)) t nil)))
  (find (place-from-2step where-type where-key)
        successful-rules
        :key #'rule-place
        :test #'place2-is-or-isin-place1))

(defun try-help-rules (places address-tokens address-i address-token)
  ;; These rules use only the known facts, not suspected values.
  (let ((places2 (remove-if-not #'certainty-rule-p places)))
    (dolist (rule (gethash address-token *geo-tokens* nil))
      (when (and (help-rule-p rule)
                 (not (member (rule-id rule) places :key #'rule-id :test
#'=))
                 (the-adj-test (1+ address-i) address-tokens (rest (first
(rule-parmdata rule)))))
        (let* ((rule-parms (rule-parmdata rule))
               (successp (try-help-rules-clauses places2
                                                 (second rule-parms)
                                                 (if (numberp (third rule-
parms))
                                                   (fourth rule-parms)
                                                   (list (third rule-parms)
(fourth rule-parms))))))
          (setq places (maybe-add-place places rule successp))
          (when successp
            (push rule places2))))))
  (values places))

(defun try-help-rules-clauses (places boolean-op wherespecs)
   (if (null wherespecs)
      (ecase boolean-op
        (-AND- t)    ; AND all -- goes until finds a false
        (-NOT- t)    ; NOT any -- goes until finds a true, returns false
        (-OR-  nil));  OR any -- goes until finds a true, returns true
      (let ((clause-result (the-help-test places (first wherespecs) (second
wherespecs))))
        (ecase boolean-op
          (-AND- (if clause-result
                     (try-help-rules-clauses places boolean-op (cddr
wherespecs))
                     (values nil)))
          (-NOT- (if clause-result
                     (values nil)
                     (try-help-rules-clauses places boolean-op (cddr
wherespecs)))))
```

```
            (-OR-  (if clause-result
                       (values t)
                       (try-help-rules-clauses places boolean-op (cddr
wherespecs)))))))))

(defun maybe-add-place (places rule addp)
   (if addp
      (cons rule places)
      places))

;;; ---------------------------------------------------------------------------
-----------|

(defun tokenize-address (address)
   ;; breaks tokens at whitespace.
   ;; dots are separate token if they are on trailing edge of other token.
   (let ((tokens (make-array 25 :adjustable t :fill-pointer 0))
         (token-in-progress (make-array 25 :adjustable t :fill-pointer 0))
         (last-char #\Space)
         c)
      (flet ((whitespace-p (chr)
               (char= #\Space chr))
             (add-to-token-in-progress (chr)
               (vector-push-extend chr token-in-progress))
             (end-of-token ()
               (when (plusp (fill-pointer token-in-progress))
                  (vector-push-extend (coerce token-in-progress 'string)
tokens)
                  (setf (fill-pointer token-in-progress) 0))))
        (dotimes (i (length address))
           (cond ((char= (setq c (aref address i)) #\Space)
                  (unless (whitespace-p last-char)
                     (end-of-token)))
                 ((find c ",." :test #'char=)
                  (if (or (whitespace-p last-char)
                          (and (< (1+ i) (length address))
                               (not (whitespace-p (aref address (1+ i))))))
                     (add-to-token-in-progress c)
                     (progn
                       (end-of-token)
                       (add-to-token-in-progress c)
                       (end-of-token))))
                 ((char= c #\/)
                  (end-of-token))
                 (t (add-to-token-in-progress c)))
           (setq last-char c))
        (unless (whitespace-p last-char)
           (end-of-token))
        ;;
        ;; Treat email addresses specially
        ;;
        (let ((last-token (vector-pop tokens))
              (trailing-period-p nil))
           (when (and (string= last-token ".")
                      (plusp (fill-pointer tokens)))
              (setq trailing-period-p t)
              (setq last-token (vector-pop tokens)))
           (multiple-value-bind (domains email)
                                (tokenize-email last-token)
             (when (null domains)
               (vector-push last-token tokens)
```

```
                ;; Preserve trailing period because "USA." needs it, for
example.
                (when trailing-period-p
                  (vector-push "." tokens)))
              (values tokens domains email))))))

(defun tokenize-email (email)
  ;; Returns the domain as a backwards list (e.g. (edu stanford)) and the
  ;; complete email address
  ;; Would be nice to split at @ sign, then split second half at dots.
  (let ((start (position #\@ email :test #'char=)))
    (when start
      (values (tokenize-email-domain (1+ start) email)
              email))))

(defun tokenize-email-domain (start email)
   ;; Domains are returned in reverse order, eg: (edu stanford cs)
   (let ((domains nil))
      (do ((right
             (position #\. email :test #'char= :start start)
             (position #\. email :test #'char= :start start)))
          ((null right)
           (push (subseq email start) domains))
        (push (subseq email start right) domains)
        (setq start (1+ right)))
      (values domains)))


@@file places

(defstruct place
  name
  key                                   ; not used except for debugging help
  number
  subkeys
  )

(defstruct (planet (:include place))
  )

(defstruct (subordinate-place (:include place))
  isin-key
  )

(defstruct (abbreviated-place (:include subordinate-place))
  abbrev
  )

(defstruct (country (:include abbreviated-place))
  )

(defstruct (stayt (:include abbreviated-place))
  )

(defstruct (unabbreviated-place (:include subordinate-place))
  )

(defstruct (region (:include unabbreviated-place))
  )

(defstruct (city (:include unabbreviated-place))
```

```
  )

(defstruct (institution (:include unabbreviated-place))
  www
  wwwdir
  (ispartof nil)
  )

(defparameter *world-key* "pl#earth")

(defvar *place-canada*)              ; set when rules read in and processed
(defvar *place-ireland*)
(defvar *place-uk*)
(defvar *place-usa*)

;; Must be synchronized with *pickfields*
(defmethod place-pfchoice-key ((p country))     'oc)
(defmethod place-pfchoice-key ((p stayt))       'ts)
(defmethod place-pfchoice-key ((p city))        'ic)
(defmethod place-pfchoice-key ((p institution)) 'ni)

(defmethod place-key-from-2step (class-abbrev place-abbrev)
  (error "Illegal class or place abbreviation: ~S ~S" class-abbrev place-
abbrev))

(defmethod place-key-from-2step ((class-abbrev (eql 'co)) place-abbrev)
  (format nil "co#~(~A~)" place-abbrev))
(defmethod place-key-from-2step ((class-abbrev (eql 'st)) place-abbrev)
  (format nil "st#~(~A~)" place-abbrev))
(defmethod place-key-from-2step ((class-abbrev (eql 'ci)) name)
  (string-downcase name))
(defmethod place-key-from-2step ((class-abbrev (eql 'in)) name)
  (string-downcase name))
(defmethod place-key-from-2step ((class-abbrev (eql 'un)) key)
  (format nil "un#~(~A~)" key))

(defun placekey= (k1 k2)
  (string-equal k1 k2))

(defvar *places*)
(defvar *place-keys*)
(defvar *world-key*)

(defun prepare-to-load-places ()
  ;; Called just before places are first loaded.
  (setf *places* (make-array 1000 :adjustable t :fill-pointer 0 :element-type
'place))
  (setf *place-keys* (make-hash-table :test #'equal :size 2000))
  (values))

(defun place-number-from-place-key (place-key &key must-find-p)
  (or (gethash place-key *place-keys* nil)
      (if must-find-p
        (error "Unable to find place key: ~S" place-key)
        nil)))

(defun place-from-place-key (place-key &key must-find-p)
  (let ((place-number (place-number-from-place-key place-key :must-find-p
must-find-p)))
    (when place-number
      (place-from-place-number place-number))))
```

```
(defun place-from-place-number (place-number)
  (aref *places* place-number))

(defun place-from-2step (class-abbrev place-abbrev)
  (place-from-place-key (place-key-from-2step class-abbrev place-abbrev)))

(defmethod place1-is-or-isin-place2 (p1 p2)
  (declare (ignore p1 p2))
  (values t))

(defmethod place1-is-or-isin-place2 ((p1 subordinate-place) p2)
  (or (eql p1 p2)
      (place1-is-or-isin-place2 (place-superplace p1) p2)))

(defmethod place2-is-or-isin-place1 (p1 p2)
  (eql p1 p2))

(defmethod place2-is-or-isin-place1 (p1 (p2 subordinate-place))
  (or (eql p1 p2)
      (place2-is-or-isin-place1 p1 (place-superplace p2))))

(defun confirm-place-key (place-key)
  (assert (place-number-from-place-key place-key :must-find-p t) nil
          "Undefined place key: ~S" place-key)
  (values place-key))

(defmethod place-superplace (place)
  (declare (ignore place))
  nil)

(defmethod place-superplace ((place subordinate-place))
   (place-from-place-key (subordinate-place-isin-key place) :must-find-p t))

(defun place-placekey= (place placekey)
  (placekey= (place-key place) placekey))

(defun top-place-p (place)
  ;; There can be only one.
  (planet-p place))

(defun top-place ()
  (place-from-place-key *world-key*))

(defun initialize-place (name rules isin-key place-creation-fn &rest
creation-args0)
  ;; Called in two ways: (1) when initial place db read in, (2) dynamically,
when
  ;; we create "unclassified" places.  Only in case 1 will there be
invisibles.
  ;; Unless it's going to be invisible, we create a new object and return it.
  (declare (dynamic-extent creation-args))
  (if (member :invisible rules)
    (progn
      (when (intersection '(:dx1 :dx2 :placekey-var) rules)
        (assert (not (find :placekey-var rules)) nil
                "Cannot use :PLACEKEY-VAR in invisible rule ~S ~S" creation-
args0 rules)
        (assert name nil
                "Cannot use :DX1 or :DX2 in invisible rule ~S ~S" creation-
args0 rules))
```

```
      (fill-in-place-rules
       (place-from-place-key isin-key)
       (cond ((member :dx1 rules) (list* :dx     name (delete :dx1 rules)))
             ((member :dx2 rules) (list* :dx-adj name (delete :dx2 rules)))
             (t                   rules)))
      (values))
    (let* ((creation-args (if isin-key
                              (list* :isin-key isin-key creation-args0)
                              creation-args0))
           (place (apply place-creation-fn creation-args))
           (key (getf creation-args :key))
           (number (fill-pointer *places*)))
      (assert (null (gethash key *place-keys* nil)) nil
              "Non-unique key ~S for place ~S" key place)
     ;(setf (place-key place) key)
      (setf (place-number place) number)
      (vector-push-extend place *places*)
      (setf (gethash key *place-keys*) number)
      (fill-in-place-rules place rules)
      (initialize-place-subkeys place)
      (values place))))

(defmethod initialize-place-subkeys ((place subordinate-place))
  ;; Pushnew used only to make things simpler in development environment.
  ;; (In case, for example, we redefine cities only.)
  ;; In production environment, could use Push.
  (pushnew (place-key place) (place-subkeys (place-superplace place)) :test
#'string=))

(defmethod initialize-place-subkeys (place)
  (declare (ignore place)))

;;; ---------------------------------------------------------------------------
-----------|

(defun make-a-planet ()
  (initialize-place nil
                    nil
                    nil
                    #'make-planet
                    :name "The World"
                    :key *world-key*))

(defun make-a-country (spec)
  (let ((abbrev (first spec))           ; a symbol
        (name   (second spec))
        (rules  (cddr spec)))
    (initialize-place name
                      (list* :tld abbrev rules) ; country abbrev = www top
level domain
                      *world-key*
                      #'make-country
                      :key (place-key-from-2step 'co abbrev)
                      :abbrev (symbol-name abbrev)
                      :name name))
  (values))


(defun make-a-state (spec)
  (let ((abbrev         (symbol-name (first spec)))
        (country-abbrev (symbol-name (second spec)))
```

```
        (name           (third spec))
        (rules          (cdddr spec)))
    (initialize-place name
                      rules
                      (confirm-place-key (place-key-from-2step 'co country-
abbrev))
                      #'make-stayt
                      :key (place-key-from-2step 'st abbrev)
                      :abbrev abbrev
                      :name name))
  (values))

(defun make-a-city (spec)
  (let ((2step-class (first  spec))    ; a symbol
        (isin-abbrev (second spec))
        (name        (third  spec))
        (rules       (cdddr  spec)))
    (assert (member 2step-class '(re st co)) nil
            "City ~S must be in a region, state, or country, not ~S"
            name 2step-class)
    (initialize-place name
                      rules
                      (confirm-place-key (place-key-from-2step 2step-class
isin-abbrev))
                      #'make-city
                      :key      (place-key-from-2step 'ci name)
                      :name     name))
  (values))

(defun make-an-institution (specs)
  (let ((name        (first specs))
        (2step-class (second specs))
        (isin        (third specs))
        (rules       (cdddr specs)))
    (if (member :invisible rules)
      (assert (eq 2step-class 'co) nil
              "Invisible institution ~S must be in a country, not ~S" name
2step-class)
      (assert (eq 2step-class 'ci) nil
              "Institution ~S must be in a city, not ~S" name 2step-class))
    (initialize-place name
                      rules
                      (confirm-place-key (place-key-from-2step 2step-class
isin))
                      #'make-institution
                      :key      (place-key-from-2step 'in name)
                      :name     name))
  (values))

(defun make-an-invisible-institution (specs)
  (let ((2step-class (first specs))
        (isin        (second specs))
        (rules       (cddr specs)))
    (make-an-institution (list* "X" 2step-class isin :invisible rules))))

(defmethod institution2-is-part-of-institution1 (i1 i2)
  (declare (ignore i1 i2))
  (values nil))

(defmethod institution2-is-part-of-institution1 ((i1 institution) (i2
institution))
```

```
  (find (place-key i1) (institution-ispartof i2) :test #'placekey=))

;;; ---------------------------------------------------------------------
-----------|

(defun find-unclassified-place (place subplace)
  (let ((key (place-key-from-2step 'un (place-key place))))
    (or (place-from-place-key key)
        (make-unclassified-place subplace place key))))

(defmethod make-unclassified-place ((subplace country) place key)
  (make-unclassified place #'make-country :key key :abbrev key))

(defmethod make-unclassified-place ((subplace stayt) place key)
  (make-unclassified place #'make-stayt :key key :abbrev key))

(defmethod make-unclassified-place ((subplace city) place key)
  (make-unclassified place #'make-city :key key))

(defmethod make-unclassified-place ((subplace institution) place key)
  (make-unclassified place #'make-institution :key key))

(defun make-unclassified (place creation-fn &rest creation-args)
  (let ((name (format nil "unclassifiable ~A" (place-name place))))
    (assert (getf creation-args :key) nil) ; for debugging only
    (apply #'initialize-place
           name nil (place-key place)
           creation-fn :name name creation-args)))


@@file platform

(defun open-text-window ()
  (values
   #-mcl (open-stream 'text-edit-window *lisp-main-window* :output)
   #+mcl (fred)))


@@file query04.htm

<html>
<head>
<script>
function do_submit() {
 var frm1 = document.forms["inputs"];
 var search = '"'  + frm1.elements["term"].value + '"'; // [All Fields])
 search += ' AND ("human"[MeSH Terms])';
 search += ' AND (1990:2000[PDAT])';
 search += ' NOT (LETTER[PT])';
 search += frm1.elements["language"].value
 search += frm1.elements["agegroup"].value
 //
 var frm2 = document.forms["marshall"];
 frm2.elements["term"].value = search;
 frm2.elements["dispmax"].value = frm1.elements["maxdisp"].value;
 //alert( search );
 frm2.submit();
}
</script>
</head>
<body>
```

```
<form name="marshall" action="http://www.ncbi.nlm.nih.gov/entrez/query.fcgi">
<input type="hidden" name="cmd" value="search">
<input type="hidden" name="db"  value="PubMed">
<input type="hidden" name="doptcmdl" value="MEDLINE">
<input type="hidden" name="term">
<input type="hidden" name="dispmax">
</form>

<form name="inputs" action="javascript:do_submit()">
<table border="0">
<tr>
<td>Search for:</td>
<td><input type="typein" name="term" size="30" value="rash"></td>
</tr>
<tr>
<td>Dispmax:</td>
<td><input type="typein" name="maxdisp" size="5" value="100"></td>
</tr>
<tr>
<td>Languages</td>
<td><select name="language">
<option value=""> All
<option value=" AND eng[LA]" selected> English
<option value=" AND fre[LA]"> French
<option value=" AND ger[LA]"> German
<option value=" AND ita[LA]"> Italian
<option value=" AND jpn[LA]"> Japanese
<option value=" AND rus[LA]"> Russian
<option value=" AND spa[LA]"> Spanish
<option value=" NOT eng[LA]"> All Non-English
</select></td>
</tr>
<tr>
<td>Age Groups:</td>
<td><select name="agegroup">
<option value="" selected> All
<option value=" AND infant, newborn [MH]"> Infant, Newborn (0 to 1 month)
<option value=" AND infant [MH:NOEXP]"> Infant (1 to 23 months)
<option value=" AND child, preschool [MH]"> Child, Preschool (2 to 5 years)
<option value=" AND child [MH:NOEXP]"> Child (6 to 12 years)
<option value=" AND adolescence [MH]"> Adolescence (13 to 18 years)
<option value=" AND child [MH]"> All Child (0 to 18 years)
<option value=" AND adult [MH:NOEXP]"> Adult (19 to 44 years)
<option value=" AND middle age [MH]"> Middle Age (45 to 64 years)
<option value=" AND aged [MH]"> Aged (65+ years)
<option value=" AND aged, 80 and over[MH]"> Aged 80 (80+ years)
<option value=" AND adult [MH]"> All Adult (19+ years)
</select></td>
</tr>
<tr>
<td colspan="2" align="center"><input type="submit"><br><input type="button"
value="Sbmt(NN)" onclick="do_submit()"></td>
</tr>
</table>
</form>
</body>
</html>

@@file resultify.htm
```

```
<html>
<head>
<title>Web of Hope</title>
</head>
<body>
<form action="http://10.1.0.4/cgi-bin/1.cgi">
<input type="hidden" name="fn" value="read_and_report_search">

<table>
<tr>
<td><b>Filename:</b></td>
<td><input type="typein" name="filename" value="brocc.html"></td>
</tr>
<tr>
<td valign="top"><b>Format:</b></td>
<td valign="top">
<input type="radio" name="format" value="geo_alpha" CHECKED>Geographical -
alphabetical
<br>
<input type="radio" name="format" value="geo_score">Geographical - by score
<br>
<input type="radio" name="format" value="byauthor">By author
<br>
<input type="radio" name="format" value="bypaper">By paper
</td>
</tr>
<tr><td> </td></tr>
<tr>
<td colspan="2" align="center"><input type="submit"></td>
</tr>
</table>

</form>
</body>
</html>


@@file rules

(defvar *geo-tokens*)
(defvar *rule-count*)
(defvar *tlds*)

(defstruct rule
  id strength method parmdata placekey placenum)

(defun rule-place (rule)
  (place-from-place-number (rule-placenum rule)))

(defun prepare-to-load-rules ()
  (setf *rule-count* 0)
  (setf *geo-tokens* (make-hash-table :size 2000 :test #'equal))
  (setf *tlds* (make-array 200 :adjustable t :fill-pointer 0))
  (values))

;;; ------------------------------------------------------------------------
-----------|

(defun add-rule (strength method place parm/s)
  (let ((rule (make-rule
               :id       (incf *rule-count*)
```

```
                :strength strength
                :method   method
                :placekey (place-key place)      ; for debugging. Should not be
used.
                :placenum (place-number place)
                :parmdata parm/s)))
    (case method
      ((:all :adj :domayn) (add-rule1 rule (first parm/s)))
      (:one                (add-rule1 rule parm/s))
      (:helpz              (add-rule1 rule (caar parm/s)))
      (t                   (error "Unknown method in rule: ~S ~S ~S ~S"
                                  method strength (place-key place)
parm/s)))))

(defun add-rule1 (rule tokenstring)
   (assert (stringp tokenstring) nil
     "Token ~S is not a string in rule ~S" tokenstring rule)
   (push rule (gethash (string-upcase tokenstring) *geo-tokens*)))

(defun certainty-rule-p (rule)
  (eql :dx (rule-strength rule)))

(defun domain-rule-p (rule)
   (eql :domayn (rule-method rule)))

(defun help-rule-p (rule)
   (eql :helpz (rule-method rule)))

(defun certainrule-place (rule)
  "Returns the place or the world."
  (if (certainty-rule-p rule)
    (rule-place rule)
    (place-from-place-key *world-key*)))

(defun fill-in-place-rules (place spec)
  (when spec
    (let* ((rule-key (first spec))
           (parms    (rest spec))
           (parm1    (first parms)))
      (fill-in-place-rules
       place
       (ecase rule-key
         (:dx1     (add-rule :dx  :one place (place-name place))
                   parms)
         (:dx      (add-rule :dx  :one place parm1)
                   (rest parms))
         (:zip     (add-rule :dx  :one place parm1)
                   (rest parms))
         (:sus     (add-rule :sus :one place parm1)
                   (rest parms))
         (:dx2     (add-rule :dx  :adj place (coerce (tokenize-address
(place-name place)) 'list))
                   parms)
         (:sus-adj (add-rule :sus :adj place (coerce (tokenize-address parm1)
'list))
                   (rest parms))
         (:dx-all  (add-rule :dx  :all place (coerce (tokenize-address parm1)
'list))
                   (rest parms))
         (:dx-adj  (add-rule :dx  :adj place (coerce (tokenize-address parm1)
'list))
```

```
                    (rest parms))

          (:dx-helpx (add-rule :dx :helpz place (list
                                                 (coerce (tokenize-address
parm1) 'list)
                                                 (second parms)
                                                 (third parms)
                                                 (fourth parms)))
                     (cddddr parms))
          (:sus-helpx (add-rule :sus :helpz place (list
                                                   (coerce (tokenize-address
parm1) 'list)
                                                   (second parms)
                                                   (third parms)
                                                   (fourth parms)))
                      (cddddr parms))

         ;;
          ;; For countries only
          ;;
          (:dx1c       (add-rule :dx  :adj place (list (place-name place)
".."))
                       parms)
          (:dx-island  (add-rule :dx :adj place (append
                                                 (coerce (tokenize-address
parm1) 'list)
                                                 (list "ISLAND")))
                       (rest parms))
          (:dx-islands (let ((tokens (coerce (tokenize-address parm1) 'list)))
                         (add-rule :dx :all place (append tokens (list
"ISLANDS")))
                         (add-rule :dx :all place (append tokens (list
"ISLAND"))))
                       (rest parms))
          (:tld        (let ((tld-string (string-upcase (symbol-name parm1))))
                         (add-rule :dx :domayn place (list tld-string))
                         (assert (not (find tld-string *tlds* :test
#'string=)) nil
                                            "Duplicate top-level domain ~S" tld-
string)
                         (vector-push-extend tld-string *tlds*))
                       (rest parms))
          ;;
          ;; For institutions only. These are not rules!
          ;;
          (:domain     (add-rule :dx :domayn place (tokenize-email-domain
                                                    0 (string-upcase parm1)))
                       (when (institution-p place)
                         (setf (institution-www place) (format nil
"http://www.~A" parm1)))
                       (rest parms))
          (:edu-domain (add-rule :dx :domayn place (list "EDU" (string-upcase
parm1)))
                       (setf (institution-www place) (format nil
"http://www.~A.edu" parm1))
                       (rest parms))
          (:www        (setf (institution-www place) parm1)
                       (rest parms))
          (:wwwdir     (setf (institution-wwwdir place) parm1)
                       (rest parms))
          (:ispartof   (push parm1 (institution-ispartof place))
```

```
                        (rest parms))

          ;; Misc. non-rules

          (:place-var (set parm1 place)
                      (rest parms))
          (:invisible parms)
          )))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; These next three functions are for debugging help only

(defun gt (key)
   (format t "~{~%~S~}~%*~%~{~%~S~}"
     (gethash key *geo-tokens*)
     (gethash (string-upcase key) *geo-tokens*)))

(defun show-geo ()
   (let ((places nil))
      (maphash #'(lambda (string values)
                   (push (cons string values) places))
        *geo-tokens*)
      (dolist (string+values (sort places #'string< :key #'first))
         (format t "~%~A~{~%  ~S~}"
           (car string+values)
           (cdr string+values)))))

(defun l (address)
   (format t "~{~%~S~%~}" (locate address)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; ae1 000728 created as new file

@@file serv_mcl

(in-package :ccl)

#|
Step by step guide to web server programming on the Macintosh with MCL.

Introduction:  Common Lisp is a great tool for building CGI applications.
A complete web server built in Common Lisp, called CL-HTTP, is available
for several computing platforms, but it is a big, complicated package
that, by all accounts, takes awhile to learn.

As a Macintosh user, I wanted a simpler, less imposing way to build CGI
applications in Macintosh Common Lisp.  SERVERGLUE.LISP does that.  This
small bit of code, combined with the freeware webservers available for
the Mac, provides the missing piece needed to start building CGI
applications in Lisp.  It is derived from code originally developed by
Michael Travers of the MIT Media Lab.

If you are new to web CGI authoring, you will want to skip over the code for
now, and read the instructions.
|#


;;;; It should respond to event by calling one of two functions:
;;;; cl-user::handle-cgi-post or cl-user::handle-cgi-get

(require :appleevent-toolkit)
```

```
(install-appleevent-handler :|WWW½| :|sdoc| 'handle-cgi-event)

(defconstant *crlf* (format nil "~C~C" #\Linefeed #\Newline))

(defparameter *http-fmt*
  (format nil "HTTP/1.0 200 OK~A~
               MIME-Version: 1.0~:*~A~
               Content-type: text/html~:*~A~:*~A~~A~:*~A"
          *crlf*))

(defparameter *http-fmt*
  (format nil "Content-type: text/html ~A~:*~A ~~A~:*~A" *crlf*))

(defparameter *http-fmt*                    ; Use this with Microsoft personal
web server
  (format nil "~~A~A" *crlf*))

(defun handle-cgi-event (application theAppleEvent reply handlerRefcon)
  (declare (ignore application handlerRefcon))
  (let ((method (ae-get-parameter-char theAppleEvent :|meth| t)))
    (ae-put-parameter-char
     reply
     #$keyDirectObject
     (format nil *http-fmt*
             (cl-user::handle-cgi (ae-get-parameter-char
                                   theAppleEvent
                                   (if (string= method "POST") :|post|
:|kfor|)
                                   t))))))

#|
(defun handle-cgi-event (application theAppleEvent reply handlerRefcon)
  "Returns all information accessible to CGI script."
  (declare (ignore application handlerRefcon))
  (ae-put-parameter-char
   reply
   #$keyDirectObject
   (format nil *http-fmt*
           (format nil "<html><body><table
border>~{~A~}</table></body></html>"
                   (mapcar #'(lambda (field)
                               (format nil "<tr><td>~S</td><td>~S</td></tr>"
                                       field
                                       (ae-get-parameter-char theAppleEvent
field t)))
                           '(:|kfor| :|user| :|pass| :|frmu| :|addr| :|post|
:|meth|
                             :|svnm| :|svpt| :|scnm| :|ctyp| :|refr| :|Agnt|
))))))
|#

;;;; End of Macintosh-specific source code.


;; End

@@file states

(mapc #'make-a-state
```

```
  '( ;; from http://new.usps.com/cgi-
 bin/uspsbv/scripts/content.jsp?A=B&D=10090&K=B&U=X&U1=B&U2=H#states
    (al us "Alabama" :dx1)
    (ak us "Alaska"  :dx1)
    (az us "Arizona" :dx1 :dx-adj "Ariz.")
    (ar us "Arkansas" :dx1)
    (ca us "California" :dx1 :dx-adj "Calif.")
    (co us "Colorado" :dx1)
    (ct us "Connecticut" :dx1)
    (de us "Delaware" :dx1)
    (dc us "District of Columbia" :dx2)
    (fl us "Florida" :dx1)
    (ga us "Georgia" :sus "Georgia" :dx-helpx "Georgia" -AND- co us)
    (hi us "Hawaii" :dx1)
    (id us "Idaho" :dx1)
    (il us "Illinois" :dx1)
    (in us "Indiana" :dx1)
    (ia us "Iowa" :dx1)
    (ks us "Kansas" :dx1)
    (ky us "Kentucky" :dx1)
    (la us "Lousiana" :dx1)
    (me us "Maine" :dx1)
    (md us "Maryland" :dx1)
    (ma us "Massachusetts" :dx1)
    (mi us "Michigan" :dx1)
    (mn us "Minnesota" :dx1)
    (ms us "Mississippi" :dx1)
    (mo us "Missouri" :dx1)
    (mt us "Montana" :dx1)
    (ne us "Nebraska" :dx1)
    (nv us "Nevada" :dx1)
    (nh us "New Hampshire" :dx2)
    (nj us "New Jersey" :dx2)
    (nm us "New Mexico" :dx2)
    (ny us "New York"   :sus-adj "New York" :dx-adj "New York, USA" :dx-adj
"NY, USA")
    (nc us "North Carolina" :dx2)
    (nd us "North Dakota"   :dx2)
    (oh us "Ohio" :dx1)
    (ok us "Oklahoma" :dx1)
    (or us "Oregon" :dx1)
    (pa us "Pennsylvania" :dx1)
    (pr us "Puerto Rico" :dx2)
    (ri us "Rhode Island" :dx2)
    (sc us "South Carolina" :dx2)
    (sd us "South Dakota"   :dx2)
    (tn us "Tennessee" :dx1)
    (tx us "Texas" :dx1)
    (ut us "Utah" :dx1)
    (vt us "Vermont" :dx1)
    (va us "Virginia" :sus-helpx "Virginia" -NOT- 2 (st wv in "Virginia Mason
Medical Center"))
    (wa us "Washington" :sus "Washington")
    (wv us "West Virginia" :dx2)
    (wi us "Wisconsin" :dx1)
    (wy us "Wyoming" :dx1)
    ;;
    ;; from: http://www.canadapost.ca/CPC2/addrm/pclookup/pccivic.shtml
    ;; ENGLISH ONLY!
    ;;
    (ab ca "Alberta" :dx1)
```

```
    (bc ca "British Columbia" :dx2)
    (mb ca "Manitoba" :dx1)
    (nb ca "New Brunswick" :dx2)
    (nf ca "Newfoundland" :dx1)
    (ns ca "Nova Scotia" :dx2)
    (nt ca "Northwest Territories and Nunavut" :dx2 :dx "Nunavut") ;;
TERRITORIES
    (on ca "Ontario" :sus "Ontario" :dx-helpx "Ontario" -AND- co ca) ;
Ontario, CA
    (pe ca "Prince Edward Island" :dx2)
    (qc ca "Quebec" :dx1)
    (sk ca "Saskatchewan" :dx1)
    (yt ca "Yukon" :dx1)
   ))

(boot::load-logical-path-file "source:city_inst")

@@file tokenize

(in-package "TOKENIZER")

(defparameter *accum-alpha-cap 0)
(defparameter *accum-alpha-low 1)
(defparameter *accum-digit 2)                    ;alpha digit #
(defparameter *whitespace 3)                     ;space /
(defparameter *commalike 4)                      ;comma
(defparameter *periodlike 5)                     ;.
(defparameter *at-sign 6)                        ;@
(defparameter *ampersand 7)                      ;&
(defparameter *paren-start 8)                    ;([{
(defparameter *paren-stop 9)                     ;)]}
(defparameter *email-xchar 10)                    ;_
(defparameter *unknown-char 11)                  ;all else

(defparameter *char-class-equal-test* #'=)

(defparameter *charclasses* (make-array 256 :initial-element *unknown-char))

(defun initialize-charclasses (classlist)
  (when classlist
    (let ((chars (first classlist))
          (code (second classlist)))
      (dotimes (i (length chars))
        (setf (aref *charclasses* (char-code (aref chars i))) code))
      (initialize-charclasses (cddr classlist)))))

(initialize-charclasses
 (list "ABCDEFGHIJKLMNOPQRSTUVWXYZ" *accum-alpha-cap
       "abcdefghijlkmnopqrstuvwxyz" *accum-alpha-low
       "0123456789" *accum-digit
       " /" *whitespace
       "," *commalike
       "." *periodlike
       "&" *ampersand
       "_" *email-xchar
       "@" *at-sign
       "({[" *paren-start
       ")}]" *paren-stop))

(defmacro charcase (selection-key &rest clauses)
   (let ((var (gensym)))
```

```
      (labels ((emit-clause (clause)
                 (list (list '= var (first clause)) (second clause))))
        `(let ((,var ,selection-key))
           (cond
             ,@(mapcar #'emit-clause clauses)
             (t (error "Unknown character class ~S encountered in
CHARCASE macro." ,var))))))))

(defun dbp (&rest args)
  (format t "~%;;;~{ ~S~}" args))

(defun char-class (c)
  (aref *charclasses* (char-code c)))

;;; Token inf
(defparameter *default-tokeninf* 0)
(defparameter *ti-starts-with-digit* (logior  1 *default-tokeninf*))
(defparameter *ti-maybe-email*       (logior  2 *default-tokeninf*))
(defparameter *ti-embedded-period*   (logior  4 *default-tokeninf*))
(defparameter *ti-ends-with-period*  (logior  8 *default-tokeninf*))
(defparameter *ti-has-comma*         (logior 16 *default-tokeninf*))
(defparameter *ti-capitalized        (logior 32 *default-tokeninf*))
(defparameter *ti-end                (logior 64 *default-tokeninf*))

(defparameter *ti-interest-ampersand*  (logior  256 *default-tokeninf*))
(defparameter *ti-interest-parenstart* (logior  512 *default-tokeninf*))
(defparameter *ti-interest-parenstop*  (logior 1024 *default-tokeninf*))
(defparameter *ti-interest-underscore* (logior 2048 *default-tokeninf*))
(defparameter *ti-interest-misc-char*  (logior 4096 *default-tokeninf*))

(defun tz (str)
   (multiple-value-bind (strs infs)
       (tokenize str)
      (dotimes (i (length strs))
         (format t "~% ~S * ~A" (aref infs i) (aref strs i)))))

(defun tokenize (str)
   (let ((len (length str))
         (last-i (1- (length str)))
         curchar)
      (multiple-value-bind (tokenstrs tokeninfs curtoken)
          (get-token-arrays-from-pool len)
         (flet ((add-token (tokenstring)
                  (vector-push-extend tokenstring tokenstrs)
                  (vector-push-extend *default-tokeninf* tokeninfs) ; prepare
for next token
                  (setf (fill-pointer curtoken) 0))
                (set-forthcoming-tokens-inf (inf-value)
                 (unless (= inf-value *default-tokeninf*)
                    (setf (aref tokeninfs (fill-pointer tokenstrs))
                          (logior (aref tokeninfs (fill-pointer tokenstrs))
                            inf-value)))))
           (macrolet ((end-current-token ()
                        `(when (plusp (length curtoken))
                           (add-token (coerce (COPY-SEQ curtoken)
'string))))
                      (accumulate-curchar ()
                        `(vector-push-extend curchar curtoken))
                      (accumulate-curchar-with-inf (inf-value)
                       `(progn
                          (set-forthcoming-tokens-inf ,inf-value)
```

```
                        (vector-push-extend curchar curtoken)))
                    (first-char-in-token-p ()
                      `(zerop (length curtoken))))
              (vector-push-extend *default-tokeninf* tokeninfs) ; prepare for
 first token
              (do ((i 0 (incf i)))
                  ((= i len))
                 (charcase (char-class (setq curchar (aref str i)))
                   (*accum-alpha-cap (if (first-char-in-token-p)
                                         (accumulate-curchar-with-inf *ti-
 capitalized)
                                         (accumulate-curchar)))
                   (*accum-alpha-low (accumulate-curchar))
                   (*accum-digit     (accumulate-curchar-with-inf (if (first-
 char-in-token-p)
                                                                      *ti-
 starts-with-digit*
                                                                      *default-
 tokeninf*)))
                   (*whitespace      (end-current-token))
                   (*commalike       (progn
                                       (end-current-token)
                                       (accumulate-curchar-with-inf *ti-has-
 comma*)
                                       (end-current-token)))
                   (*periodlike      (if (next-char-ends-token-p str i last-i)
                                         (accumulate-curchar-with-inf *ti-ends-
 with-period*)
                                         (accumulate-curchar-with-inf *ti-
 embedded-period*)))
                   (*at-sign         (accumulate-curchar-with-inf *ti-maybe-
 email*))
                   (*ampersand       (if (and (first-char-in-token-p)
                                              (next-char-ends-token-p str i
 last-i))
                                         (add-token "and")
                                         (accumulate-curchar-with-inf *ti-
 interest-ampersand*)))
                   (*paren-start     (if (next-char-ends-token-p str i last-i)
                                         (accumulate-curchar-with-inf *ti-
 interest-parenstart*)
                                         (set-forthcoming-tokens-inf *ti-
 interest-parenstart*)))
                   (*paren-stop      (if (next-char-ends-token-p str i last-i)
                                         (accumulate-curchar-with-inf *ti-
 interest-parenstop*)
                                         (set-forthcoming-tokens-inf *ti-
 interest-parenstop*)))
                   (*email-xchar     (accumulate-curchar-with-inf *ti-interest-
 underscore*))
                   (*unknown-char    (accumulate-curchar-with-inf *ti-interest-
 misc-char*))))
              (end-current-token)
              (set-forthcoming-tokens-inf *ti-end)
              (add-token "*END*")
              (values tokenstrs tokeninfs))))))

#|
(defun get-token-arrays-from-pool (max)
  ;; Returns buffer arrays guaranteed to handle a line of <max> characters.
  ;; This version does not get from pool, obviously.
```

```
   (values (make-array 30 :adjustable t :fill-pointer 0 :element-type 'string)
           (make-array 30 :adjustable t :fill-pointer 0 :element-type
'integer)
           (make-array 25 :adjustable t :fill-pointer 0 :element-type
'character)))
|#

(cl-user::defresource tokenstr-buffer :initial-copies 5
  :constructor (make-array 30 :adjustable t :fill-pointer 0 :element-type
'string))
(cl-user::defresource tokeninf-buffer :initial-copies 5
  :constructor (make-array 30 :adjustable t :fill-pointer 0 :element-type
'integer))
(cl-user::defresource curtoken-buffer :initial-copies 5
  :constructor (make-array 25 :adjustable t :fill-pointer 0 :element-type
'character))

(defun get-token-arrays-from-pool (max)
  ;; Returns buffer arrays guaranteed to handle a line of <max> characters.
  ;; This version does not get from pool, obviously.
  (values (allocate-tokenstr-buffer)
          (allocate-tokeninf-buffer)
          (allocate-curtoken-buffer)))

(let ((ending-classes (vector *whitespace *commalike)))
   (defun next-char-ends-token-p (str i last-i)
      (or (= i last-i)
          (find (char-class (aref str i)) ending-classes
            :test *char-class-equal-test*))))


@@file xserve-macros

;;; ---------------------------------------------------------------------------
-----------|
;;; CGI MACROS

(defparameter *approved-functions* nil)

(defun approved-function-p (function-symbol)
  (member function-symbol *approved-functions* :test #'eq))

(defun log-cgi-fn (fn-symbol)
  (export fn-symbol)
  (pushnew fn-symbol *approved-functions*)
  (values fn-symbol))

(export '(thread)) ; so we can use THREAD var in DEFCGIFNs in other packages.

(defmacro defcgifn (name arglist &body body)
  "To specify object in arglist, comply with !objname-key literal"
  ;; Don't know why the following expands arglist incorrectly.
  ;; `(first (pushnew (defun ,name (thread &key ,@arglist) etc.]
  `(log-cgi-fn (defun ,name ,(list* 'thread '&key arglist)
                 #+mcl (declare (ignore-if-unused thread))
                 ,@body)))

(defmacro defcgifn2 (name arglist &body body)
  ;; Would be nice to do remf calls to remove enumerated args from total
list.
  (assert (eq '&all (nth (- (length arglist) 2) arglist)) nil
```

```
          "Arglist for cgifn2 ~S lacks &all: ~S" name arglist)
  (let ((ovar (nth (1- (length arglist)) arglist))
        (keylist (butlast arglist 2)))
    `(log-cgi-fn (defun ,name ,(append (list 'thread '&rest ovar '&key)
                                       keylist
                                       '(&allow-other-keys))
                   #+mcl (declare (ignore-if-unused thread))
                   ,@body))))

;; Example:
;; (defcgifn prez (president state party)
;;   (format nil "<html><body>~A ~A ~A</body></html>" president state party))

(defparameter *htm-functions* nil)

(defmacro defhtmfn (function-name arglist propskey props &body body)
  (assert (eq 'thread (first arglist)) nil
          "First argument for a defhtmfn must be 'thread' -- (defhtmfn ~S ~S
...)"
          function-name arglist)
  (assert (eq propskey :props) nil
          "Must specify :props for defhtmfn ~S" function-name)
  ;; Now look at top level, or one level below, for a WITH-NEW-PAGE macro.
  ;; The < listp > test is needed to screen out the LET, PROGN, etc.
  (assert (or (member 'with-new-page body :key #'first)
              (member 'with-new-page (first body) :key #'(lambda (form)
                                                           (and (listp form)
                                                                (first
form)))))
          nil
          "Expect WITH-NEW-PAGE somewhere in defhtmfn ~S" function-name)
  (pushnew (cons function-name props) *htm-functions* :key #'car :test #'eq)
  `(defun ,function-name ,arglist
     (session-hx-add thread (list :HTML-GEN (quote ,function-name) ,@(rest
arglist)))
     ;;
     ;; < locally > needed in case there are declarations in the function
body
     ;; (as happens in < search-results >, for instance).
     ;;
     (throw :output-html-page (locally ,@body))))

(defmacro defaccfn (name arglist &body body)
  `(defun ,name ,(list* 'thread 'stream arglist)
     #+mcl (declare (ignore-if-unused thread stream))
     ,@body))

;;; ---------------------------------------------------------------------------
-----------|
;;; THREAD STREAM MACROS

(defmacro with-thread-output (streamvar thread &body body)
  `(let ((,streamvar (thread-accumulator-stream ,thread)))
     ,@body))

(defmacro formatt (thread fmt$ &rest args)
  (declare (dynamic-extent args))
  `(format (thread-accumulator-stream ,thread) ,fmt$ ,@args))

(defun formatt-fn (thread fmt$ &rest args)
  (format (thread-accumulator-stream thread) fmt$ args))
```

```
(defmacro with-thread-output ((var thread) &body body)
  `(let ((,var (thread-accumulator-stream ,thread)))
     ,@body))

(defmacro terprit (thread)
  `(terpri (thread-accumulator-stream ,thread)))

(defmacro princt (value thread)
  `(princ ,value (thread-accumulator-stream ,thread)))

(defmacro with-new-page ((thred &key
                                (title nil)
                                (head "")       ; remainder of HEAD section
                                (body "")       ; attributes for BODY tag
                                (home-p t))
                         &body bodyx)
  (let ((streamvar (gensym)))
    `(with-output-to-string (,streamvar)
       (setf (thread-accumulator-stream ,thred) ,streamvar)
       (unwind-protect
         (progn
           (with-new-page1 ,thred ,title ,head ,body ,home-p)
           ,@bodyx
           (with-new-page9 ,thred)
           ,streamvar)
         (setf (thread-accumulator-stream ,thred) nil)))))

(defun with-new-page1 (thread title head-section body-tag-attributes home-p)
  (let ((actual-title (or title "-no title-")))
    (formatt thread
             "<html><head><title>~A</title>~A~%</head>~%<body
bgcolor='#ffffff' ~A>~
              <font face='sans-serif'>"
             actual-title
             head-section
             body-tag-attributes)
    (cgi-anchor thread 'MAIN)
    (html-image-tag thread "ct.gif" :align "right" :border "0")
    (princt "</a>" thread)
    ;(html-image-tag thread "betz.gif" :align "left")
    (unless home-p
      ;; put out clickable link to home page
      )
    (formatt thread "~%<center><h2>~A</h2></center>" actual-title)))

(defun with-new-page9 (thread)
  (declare (ignore wkey))
  (let ((sep$ "   |   "))
    (formatt thread "~%<p><hr><center>~%")
    (cgi-anchor-with-text thread "Home" 'homeh)
    (princt sep$ thread)
    (cgi-anchor-with-text thread "Search [broken]" 'searchh)
    (princt sep$ thread)
    (cgi-anchor-with-text thread "People" 'peopleh)
    (princt sep$ thread)
    (princt "Logout" thread)
    (princt sep$ thread)
    (princt "About us" thread)
    (princt sep$ thread)
    (princt "Comments" thread)
```

```
    (princt "</center></font></body></html>" thread)))

#|
(defmacro with-form-accumulator ((thread fn) &body body)
  (let ((streamvar (gensym)))
    `(let ((,streamvar (thread-accumulator-stream ,thread)))
       (:@@format ,thread ,streamvar "~%<form>~%" ,fn ) ; chopped down
       ,@(wwap2 thread streamvar body)
       (:@@string ,thread ,streamvar "</form>"))))
|#

@@file xserve

(in-package :cl-user)

(defmacro with-new-thread ((var session) &body body)
  ;; USE ONLY IN THIS FILE!
  `(let ((,var (make-a-thread ,session)))
     (unwind-protect
       (progn
         ,@body)
       (dispose-thread ,var))))

(defun handle-cgi (argstring)
  (let* ((keylist (tokenize-cgi-args argstring))
         (function (getf keylist :fn)))
    (if (and function
             (setq function (read-from-string function))
             (approved-function-p function))
      (progn
        (with-new-thread (thread (or (get-session keylist)          ; uses
:sn argument from keylist
                                     (make-a-session)))
          (remf keylist :fn)
          (remf keylist :sn)
          (session-hx-add thread (list* :cgi-request function keylist))
          (catch :output-html-page
            (apply function thread keylist))))
      (format nil "<html><body><h1>CGI Error: Unapproved function</h1>~A<hr>~
                   This is not an error a user can fix.  Contact: [insert
name here]<hr>
                   <h2>Server received:</h2>~A~
                   <h2>Which parses as:</h2>~
                   <table border>~{<tr><td>~S</td><td>~S</td></tr>~}</table>~
                   <p>## End ##</body></html>"
              function argstring keylist))))

;;; ---------------------------------------------------------------------------
-----------|

(defresource cgi-tokenize-buffer
  :constructor (make-array 400 :fill-pointer t :adjustable t)
  :initial-copies 1)

(defmacro with-parse-buf ((var) &body body)
  `(let ((,var (allocate-cgi-tokenize-buffer)))
     (unwind-protect
       (progn
         (setf (fill-pointer ,var) 0)
         ,@body)
       (deallocate-cgi-tokenize-buffer ,var))))
```

```
(defun tokenize-cgi-args (argstring)
  "Converts HTTP CGI arg syntax into LISP keywords-and-values list.
   Will fail if the name of a field begins with a space."
  ;; Example: fn=vote&prez=washington&vp=adams       -->
  ;;          (:fn "vote" :prez "washington" :vp "adams")
  (with-parse-buf (buf)
    (vector-push #\( buf)
    (vector-push #\: buf)
    (unwind-protect
      (let ((len (length argstring))
            c)
        (setq *read-base* 16)
        (do ((i 0))
            ((>= i len))
          (setq c (aref argstring i))
          (incf i
                (case c
                  (#\=
                   (vector-push-extend #\Space buf)
                   (vector-push-extend #\" buf)
                   1)
                  (#\&
                   (vector-push-extend #\" buf)
                   (vector-push-extend #\Space buf)
                   (vector-push-extend #\: buf)
                   1)
                  (#\+
                   (vector-push-extend #\Space buf)
                   1)
                  (#\%
                   (setq c (code-char (read-from-string argstring nil nil
:start (1+ i) :end (+ i 3))))
                   (when (char= c #\")
                     (vector-push-extend #\\ buf))
                   (vector-push-extend c buf)
                   3)
                  (t
                   (vector-push-extend c buf)
                   1)))))
      (setq *read-base* 10))            ; is cleanup form for unwind-protect
    (vector-push-extend #\" buf)
    (vector-push-extend #\) buf)
    (read-from-string (coerce buf 'string))))

;;; ------------------------------------------------------------------------
-----------|
;;; ------------------------------------------------------------------------
-----------|
;;; Edit this this value as appropriate
;;; ------------------------------------------------------------------------
-----------|
;;; ------------------------------------------------------------------------
----------|
(defparameter *cgi* "1.cgi")
;;; ------------------------------------------------------------------------
-----------|
;;; ------------------------------------------------------------------------
-----------|

(defun cgi-anchor (thread function &rest field+value-list)
```

```
  "< field+value > elements should be symbols, not strings."
  ;; DEPRECATED IN FAVOR OF CGI-ANCHOR-WITH-TEXT
  (declare (dynamic-extent field+value))
  (formatt thread
           "<a href='~A?fn=~S&sn=~S~{&~A=~A~}'>"
           *cgi*
           function
           (thread-session-key thread)
           field+value-list))

(defun cgi-anchor2 (thread function &rest field+value-list)
  ;; Outputs to string, not thread stream.
  (declare (dynamic-extent field+value))
  (format nil
          "<a href='~A?fn=~S&sn=~S~{&~A=~A~}'>"
          *cgi*
          function
          (thread-session-key thread)
          field+value-list))
#|
(defparameter *puiw*
  ;; Giving clock-time as the name (second) argument to window.open is good.
  ;; If every puiw had same name, Netscape4 would not bring a newly opened
  ;; puiw to the front.
  (format nil "<script>function puiw(fn,sn,obj) {~
               var url='~A?fn=' + fn + '&sn=' + sn + '&~S=' + escape(obj);~
               var d=new Date();~

window.open(url,d.getTime(),'height=300,width=300,scrollbars',true);~
               }</script>"
          *cgi* !objname-key))
|#
(defun cgi-anchor-puiw (thread objname fn text &key (pre "") (post ""))
  ;; puiw = pop-up information window
  (assert (stringp objname) nil "Expected a string instead of ~S" objname)
  (formatt thread "~A<a href=\"javascript:puiw('~S','~S','~A')\">~A</a>~A"
           pre fn (thread-session-key thread) objname text post))

(defun cgi-anchor-with-text (thread text function &rest field+value-list)
  "< field+value > elements should be symbols, not strings."
  (declare (dynamic-extent field+value))
  (formatt thread
           "<a href='~A?fn=~S&sn=~S~{&~A=~A~}'>~A</a>"
           *cgi*
           function
           (thread-session-key thread)
           field+value-list
           text))

#| Example of calling cgi-anchor-with-text to build link around object.
(defun cgi-anchor-text2 (thread pre text u-obj post &key fn)
  (princt pre thread)
  (cgi-anchor-with-text thread text fn !objname-key u-obj)
  (princt post thread))
|#

(defun cgi-href-start (thread function)
  "Good style suggests this should probably be deleted someday."
  (declare (dynamic-extent field+value))
  (formatt thread
           "~A?fn=~S&sn=~S"
```

```
            *cgi*
            function
            (thread-session-key thread)))

#| BUGGY!
(defun cgi-form (thread function-symbol &rest attribute+value-list)
  "< attribute+value > elements should be symbol and string, respectively."
  (declare (dynamic-extent attribute+value))
  #.(format *standard-output* "~%;;; Change form method GET to POST someday
in < cgi-form >")
  ;; Compile-time message above reminds us that GET is deprecated in HTML4,
  ;; but GET is more useful for debugging.
  (formatt thread
           #.(concatenate 'string "~%<form action='" *cgi* "' method=GET~{
~S=~S~}>")
           attribute+value-list)
  (formatt thread "~%<input type=hidden name='fn' value='~S'>~%<input
type=hidden name='sn' value=~S>~%"
           function-symbol
           (thread-session-key thread))
  )
|#

(defun cgi-form-start (thread function-symbol &rest attribute+value-list)
  "< attribute+value > elements should be symbol and string, respectively."
  (declare (dynamic-extent attribute+value))
  ;;
  ;; Compile-time message above reminds us that GET is deprecated in HTML4,
  ;; but GET is more useful for debugging.
  ;;
  #.(format *standard-output* "~%;;; Change form method GET to POST someday
in < cgi-form >")
  ;;
  (formatt thread "~%<form action='~A' method=GET~{ ~S=~S~}>" *cgi*
attribute+value-list)
  (cgi-form-hidden thread 'fn (symbol-name function-symbol) 'sn (thread-
session-key thread)))

(defun cgi-form-hidden (thread &rest attribute+value-list)
  "< attribute+value > elements should be symbol and string, respectively."
  (declare (dynamic-extent attribute+value))
  (formatt thread "~{~%<input type=hidden name='~S' value=~S>~}"
           attribute+value-list))

;;; ---------------------------------------------------------------------------
-----------|

(defparameter *sessions* nil)

(defstruct session
  key
  incept
  uuid                                  ; unique user id
  hx
  ;;
  ;; ****************************************
  ;; The application can add other slots here.
  ;; ****************************************
  ;;
  curtallykey                           ; key to current tally object
  )
```

```
(defstruct thread
  session
  incept
  accumulator-stream
  )

(defmethod print-object ((x thread) stream)
  (if t
    (format stream "<THREAD ~S>" (thread-incept x))
    (call-next-method x stream)))

(defresource vectorstack :constructor (make-array 10 :fill-pointer 0
:adjustable t))

(defun get-session (keylist)
  "Returns session object or NIL"
  (let ((key (getf keylist :sn)))
    (when key
      (find (read-from-string key) *sessions* :key #'session-key :test
#'=))))

(LET ((N 0))
  (defun make-a-session ()
    (first (push (make-session :key (random 65535)
                               :UUID (INCF N)    ; SHOULD COME FROM DATABASE
MACHINE
                               :incept (get-universal-time))
                 *sessions*))))

(defun session-hx-add (thread data)
  (let ((session (thread-session thread)))
    (push (cons (- (get-universal-time) (session-incept session))
                data)
          (session-hx session))))

(defun make-a-thread (session)
  (let ((vstack (allocate-vectorstack)))
    (setf (fill-pointer vstack) 0)
    (make-thread :session session
                 :incept (get-universal-time))))

(DEFUN DISPOSE-THREAD (THREAD)
  ;(DEALLOCATE-VECTORSTACK (THREAD-FINDINGS-TRYING-TO-INFER THREAD))
  THREAD)

(defun thread-session-key (thread)
  (session-key (thread-session thread)))
```